The Pragmatic Programmers

TURING 图灵程序设计丛书

Seven Concurrency Models in Seven Weeks

When Threads Unravel

七周七并发模型

借助Java、Go等多种语言的特长，
深度剖析所有主流并发编程模型



【美】Paul Butcher　著

黄炎　译


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Paul Butcher

资深程序员，涉猎广泛，从单片机编码到高级声明式编程无所不精。Paul是一位少年天才，8岁时就已经开始在8位机上编写游戏。最近几年他开始痴迷于赛车，想要去叫板汉密尔顿。除本书外，还著有在亚马逊获得全五星好评的《软件调试修炼之道》。

黄炎

现供职于上海爱可生，从事数据库高可用软件开发。简介如代码一样简洁。

TURING 图灵程序设计丛书

Seven Concurrency Models in Seven Weeks

When Threads Unravel

七周七并发模型



【美】Paul Butcher 著

黄炎 译

人民邮电出版社

北京

图书在版编目（C I P）数据

七周七并发模型 / （美）布彻（Butcher,P.）著 ；
黄炎译. -- 北京 : 人民邮电出版社, 2015.4
（图灵程序设计丛书）
ISBN 978-7-115-38606-9

Ⅰ. ①七… Ⅱ. ①布… ②黄… Ⅲ. ①并行程序一程
序设计 Ⅳ. ①TP311.11

中国版本图书馆CIP数据核字(2015)第038645号

内 容 提 要

并发编程近年逐渐热起来，Go 等并发语言也对并发编程提供了良好的支持，使得并发这个话题受到越来越多人的关注。本书延续了《七周七语言》的写作风格，通过以下七个精选的模型帮助读者了解并发领域的轮廓：线程与锁，函数式编程，Clojure，actor，通信顺序进程，数据级并行，Lambda 架构。书中每一章都设计成三天的阅读量。每天阅读结束都会有相关练习，巩固并扩展当天的知识。每一章均有复习，用于概括本章模型的优点和缺陷。

本书适合所有想了解并发的程序员。

◆ 著　　　[美] Paul Butcher
译　　　黄　炎
责任编辑　朱　巍
执行编辑　陈婷婷
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：15.25
字数：366千字　　　　　2015年 4 月第 1 版
印数：1－4 000册　　　　2015年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-6027号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

译　者　序

作者认为写作本书非常不易，译者深以为然。翻译本书的主要困难源自于我们与作者在知识积累上的巨大差距，作者在前言中写道“我从 1989 年开始攻读博士学位，在并行计算和分布式计算的领域深造”，那年我只有一岁。我无法精通本书介绍的七个模型中的所有技术细节，因此没有自信确保全无差错，只能竭尽所能保证译文不会有大的偏差。

在此要向提供帮助的人们致以谢意。首先，感谢图灵的编辑老师，他们辛勤的工作完善了本书的每个细节；其次，要感谢我的父亲——大连海事大学的黄映辉教授，他为本书进行了三次审校，对字句进行了细致斟酌，大幅提升了本书的可读性；然后，要感谢我的挚友——工作于 Fox News Digital 的孙培源，他为本书进行了中英文的对照审校，帮助矫正了翻译过程中的很多谬误；还要感谢工作于喜马拉雅的柳飞提供的莫大帮助；最后，要感谢这个时代，让我们有机会能参与到这个伟大的丛书系列中。

本书介绍了七种并发模型，行文通俗易懂，有数量充足且设计精良的样例来帮助读者理解。读完本书，我最大的感受是世界变得更大了，想要学习的有趣的东西变得更多了。希望大家读完后也有类似有趣的体验。

用一个亲身经历的趣事来结束本序。

几年前，我去某软公司应聘，电话面试中与面试官有如下一番对话。

面试官：你了解多线程并发吗？

我：不了解，我之前做业务系统，多线程很大程度上都是委托给容器的……

面试官：我理解了。你不太熟悉并发是吗？

我：是的。

面试官：那我们还是来聊一聊并发吧。

祝大家线程安全。

黄炎
2014 年 12 月 31 日

推 荐 序

本书将讲述一个完整的故事。

将此作为一本书的首要定位似乎有点奇怪，但对我而言这很重要。我们曾回绝数十位申请撰写“七周系列丛书”的作者，他们认为只要将七个分散主题拼凑起来就是一本书，但这有违我们的初衷。

先前的《七周七语言：理解多种编程范型》[1]讲述了一个面向对象编程语言的故事，这是很适应当时的环境的。但在多核架构的驱动下，软件复杂度的增长和并发技术的发展所带来的压力，将函数式编程推到舞台之上，并对今后的编程方式有着深远的影响。Paul Butcher 是《七周七语言》最给力的审校者之一，相识四年后，我开始理解其中原因。

Paul 一直奋斗在将高可扩展的并发技术应用于实际业务系统的第一线。读过《七周七语言》后，对于他所处的日益重要但日趋复杂的问题领域，Paul 觉得可以从编程语言级别获得一些启发。几年后，Paul 表示要写一本自己的书。他解释道：尽管编程语言在整个故事中有着重要的作用，但也只触及了问题的表面。他要为读者讲述一个更完整的故事，为非专业人士介绍现代应用程序用以解决大型并行问题的扩展性良好的重要工具。

一开始我们是持怀疑态度的。这类书是很难写的——比起其他领域的书，这类书需要花费更长的时间，而且失败的几率很高——Paul 显然选择了一块难啃的骨头。作为一个团队，我们不断磨合前进，终于从最初的大纲中研磨出一个优秀的故事。随着书稿逐渐完成，我们更加自信于 Paul 的技术能力和攻关热情。现在，我们已经确信这是一本特别的书，而且恰逢其时。随着阅读的深入，我相信你也会同意这个观点。

当你在开篇阅读到“线程与锁”这种当今最广泛使用的并发解决方案时，可能会不以为然。不过你很快就会看到这种解决方案的不足之处，并开始思考如何解决。Paul 将引领你学习多种非常不同的技术，从一些社交平台使用的 Lambda 架构，到现今世界上许多最大最可靠的电信系统使用的 actor 模型。你会学到职业高手使用的一些语言，从 Java 到 Clojure，再到基于 Erlang 的闪亮新秀 Elixir。旅途中的每一步，Paul 都将从专业的角度为你剖析其中的玄妙和精彩。

在此，我诚意奉上《七周七并发模型》。希望你和我一样乐享其中。

Bruce A. Tate
icanmakeitbetter.com 网站 CTO，七周系列丛书主编
于美国德克萨斯州奥斯汀

---

① 本书中文版电子书在图灵社区有售：http://www.ituring.com.cn/book/829。——编者注

前　　言

我从 1989 年开始攻读博士学位，在并行计算和分布式计算的领域深造，当时我便深信并发编程将成为主流。二十年后，我的观点终于得以验证——整个世界都在讨论多核以及如何发挥其优势。

学习并发不仅是为了利用多核来获得更好的性能。若正确使用并发，我们还能在程序的响应性、容错性、效率和简洁程度上获得大幅提升。

关于本书

本书延续了 Pragmatic Bookshelf 的七周系列丛书（《七周七语言》《七周七数据库》[①]《七周七网络框架》）的架构，通过七个精选的模型帮助读者了解并发领域的轮廓。这些模型中，一些已经成为主流，一些很快会成为主流，另一些虽难以成为主流，但在特定领域会威力无穷。当面对一个并发问题时，你可以借助本书准确选择合适的工具，这就是我的期望所在。

本书的每一章都设计成三天的阅读量。每天阅读结束都会有相关练习，巩固并扩展当天的知识。每一章均有复习，用于概括本章模型的优点和缺陷。

尽管有少量具有哲学意味的讨论，但本书还是侧重于实践。我强烈建议你在阅读样例时能亲手实践一下——没什么比代码更有说服力了。

本书未涉及的内容

本书不是语言参考手册。我们会使用一些较新的语言，例如 Elixir 和 Clojure，但本书关注的是并发而不是编程语言，所以不会深入介绍这些语言的具体特性。希望你通过上下文可以初步了解这些语言的主要特性，如果要对其深入探究以期充分理解，就得依靠自身的努力了。阅读本书时，如果手边开着浏览器可随时查阅语言参考手册，就会事半功倍。

本书不是安装配置手册。要运行本书的配套代码，就需要安装和运行相应工具——配套代码的 README 文件会给出一些提示，但还是要依靠你自己。本书所有的样例都采用主流工具编写，如果遇到困难，你可以在网络上找到许多帮助资料。

本书也不是面面俱到——无法囊括所有议题的每个细节。对于某些议题，本书会一笔带过或

① 本书中文版电子书在图灵社区有售：“http://www.ituring.com.cn/book/1369。——编者注

者根本不予讨论。在某些章节中，我会特意使用一些不规范的代码，目的是便于不熟悉该语言的读者来理解代码。如果你有意深入学习本书中的某种技术，建议阅读本书所提及的权威文献。

样例代码

本书讨论的所有样例都可以从本书的网站[①]下载。每个样例都包括源码和构建系统。对于每一种语言，本书都选用最通用的构建系统（Java 使用 Maven，Clojure 使用 Leiningen，Elixir 使用 Mix，Scala 使用 sbt，C 使用 GNU Make）。

大多数情况下，构建系统不仅会编译代码，而且会下载所需的额外依赖。sbt 和 Leiningen 甚至会下载对应版本的 Scala 和 Clojure 的编译器，所以你只需要下载并安装构建系统即可（在网络上可以找到详尽的安装步骤）。

不过第 7 章中使用的 C 代码是个特例，需要根据你的操作系统和显卡类型安装相应的 OpenCL 工具包（除非你使用的是 Mac，因为 Xcode 会搞定一切）。

给 IDE 用户的建议

本书使用的构建系统都在命令行下测试通过。如果你是成熟的 IDE 用户，一定知道如何将构建系统导入到 IDE 中——大多数 IDE 都会兼容 Maven，主流 IDE 也都有兼容 sbt 和 Leiningen 的插件。不过我没有在 IDE 中测试过，所以你与我一样使用命令行也许会容易一些。

给 Windows 用户的建议

所有的样例均在 OS X 和 Linux 上测试通过。理论上，它们也可以在 Windows 上运行良好，但我并没有验证过。

第 7 章中使用的 C 代码是个特例，其使用了 GNU Make 和 GCC。理论上是能被迁移到 Visual C++中，但我没有尝试过这种可能。

在线资源

本书中的样例都可以在本书网站上找到。如果你要提交勘误或者给出建议，也可以在网站上找到勘误表格和交流论坛。

Paul Butcher
Ten Tenths Consulting
paul@tententhsconsulting.com
2014 年 6 月于英国剑桥

① http://pragprog.com/book/pb7con

致　谢

当我宣告决定写本书时，一个朋友提醒道："你是不是已经忘记写第一本书时的艰辛了？"我当时一定是太天真，误认为写第二本书会容易一些。现在想来，如果不参与七周系列丛书，而是选择容易一些的题材，我的日子会好过很多。

若没有主编 Bruce Tate 和策划编辑 Jackie Carter 的鼎力相助，本书定然无法完成。感谢两位在本书写作过程中的大力支持，同时感谢 Dave 和 Andy 让我有幸参与到这个伟大的系列丛书中。

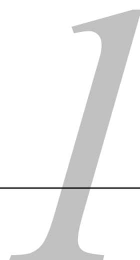
感谢许多朋友在成书早期提供的建议和反馈，他们是（排名不分先后）：Simon Hardy-Francis、Sam Halliday、Mike Smith、Neil Eccles、Matthew Rudy Jacobs、Joe Osborne、Dave Strauss、Derek Law、Frederick Cheung、Hugo Tyson、Paul Gregory、Stephen Spencer、Alex Nixon、Ben Coppin、Kit Smithers、Andrew Eacott、Freeland Abbott、James Aley、Matthew Wilson、Simon Dobson、Doug Orr、Jonas Bonér、Stu Halloway、Rich Morin、David Whittaker、Bo Rydberg、Jake Goulding、Ari Gold、Juan Manuel Gimeno Illa、Steve Bassett、Norberto Ortigoza、Luciano Ramalho、Siva Jayaraman、Shaun Parry、Joel VanderWerf。

感谢本书的技术审校者（排名不分先后）：Carlos Sessa、Danny Woods、Venkat Subramaniam、Simon Wood、Páidí Creed、Ian Roughley、Andrew Thomson、Andrew Haley、Sean Ellis、Geoffrey Clements、Loren Sands-Ramshaw、Paul Hudson。

最后，要向我的朋友、同事、家人致以谢忱和歉意。感谢你们的支持和鼓励，也请原谅过去18个月中我的种种偏执。

目　　录

第 1 章 概　　述



并发编程的概念并不新，却直到最近才火起来。一些编程语言，如Erlang、Haskell、Go、Scala、Clojure，也因对并发编程提供了良好的支持，而受到广泛关注。

并发编程复兴的主要驱动力来自于所谓的“多核危机”。正如摩尔定律[①]所预言的那样，芯片性能仍在不断提高，CPU的速度会继续提升，但计算机的发展方向已然转向多核化[②]。

Herb Sutter曾经说过：“免费午餐的时代已然终结。”[③]为了让代码运行得更快，单纯依靠更快的硬件已无法满足要求，我们需要利用多核，也就是发掘并行执行的潜力。

1.1 并发还是并行？

本书的主题是“并发”，那么又为何涉及了“并行”呢？虽然两者有所关联又常被混淆，但并发和并行的含义却是不同的。

一字之差也是差

并发程序含有多个逻辑上的独立执行块[④]，它们可以独立地并行执行，也可以串行执行。

并行程序解决问题的速度往往比串行程序快得多，因为其可以同时执行整个任务的多个部分。并行程序可能有多个独立执行块，也可能仅有一个。

我们还可以从另一种角度来看待并发和并行之间的差异：并发是问题域中的概念——程序需

① http://en.wikipedia.org/wiki/Moore%27s_law

② 作者在本章不断使用“core”“CPU”“processor”，译者在此尊重原文分别翻译成“核”“CPU”“处理器”。但译者认为此处指的都是广义的处理单元，而不是狭义的硬件。——译者注

③ http://www.gotw.ca/publications/concurrency-ddj.htm

④ 原文是“logical threads of control”，直译为“控制逻辑线程”，但在此语境下“控制”或“线程”指的并不是我们常见的“控制”和“线程”。为便于理解，在此将其译成“独立执行块”，这个概念来自于Google IO 2012的演讲“Go concurrency patterns”中引用的文档“Concurrency is not Parallelism”（http://tinyurl.com/goconcnotpar），其将这个概念称为“independently executing processes”。——译者注

要被设计成能够处理多个同时（或者几乎同时）发生的事件；而并行则是方法域中的概念——通过将问题中的多个部分并行执行，来加速解决问题。

引用Rob Pike的经典描述[①]：

并发是同一时间应对（dealing with）多件事情的能力；

并行是同一时间动手做（doing）多件事情的能力。

那么这本书讲述的是并发还是并行？



小乔爱问：

并发？并行？

我妻子是一位教师。与众多教师一样，她极其善于处理多个任务。她虽然每次只能做一件事，但可以同时处理多个任务。比如，在听一位学生朗读的时候，她可以暂停学生的朗读，以维持课堂秩序，或者回答学生的问题。这是并发，但并不并行（因为仅有她一个人，某一时刻只能进行一件事）。

但如果还有一位助教，则她们中一位可以聆听朗读，而同时另一位可以回答问题。这种方式既是并发，也是并行。

假设班级设计了自己的贺卡并要批量制作。一种方法是让每位学生制作五枚贺卡。这种方法是并行，而（从整体看）不是并发，因为这个过程整体来说只有一个任务。

超越串行编程模型

并发和并行的共同点就是它们比传统的串行编程模型更优秀。 本书将同时涵盖并发和并行（学究可能会给这本书起名为“七周七并发模型和并行模型”， 不过那样的话，封面会变得很难看）。

并发和并行经常被混淆的原因之一是，传统的“线程与锁”模型并没有显式支持并行。如果要用线程与锁模型为多核进行开发，唯一的选择就是写一个并发的程序，并行地运行在多核上。

然而，并发程序的执行通常是不确定的，它会随着事件时序的改变而给出不同的结果。对于真正的并发程序，不确定性是其与生俱来且伴随始终的属性。与之相反，并行程序可能是确定的——例如，要将数组中的每个数都加倍，一种做法是将数组分为两部分并把它们分别交给一个核处理，

① http://concur.rspace.googlecode.com/hg/talk/concur.html

这种做法的运行结果是确定的。用支持并行的编程语言可以写出并行程序，而不引入不确定性。

1.2　并行架构

人们通常认为并行等同于多核，但现代计算机在不同层次上都使用了并行技术。比如说，单核的运行速度现今仍能每年不断提升的原因是：单核包含的晶体管数量，如同摩尔定律预测的那样变得越来越多，而单核在位级和指令级两个层次上都能够并行地使用这些晶体管资源。

位级（bit-level）并行

为什么32位计算机的运行速度比8位计算机更快？因为并行。对于两个32位数的加法，8位计算机必须进行多次8位计算，而32位计算机可以一步完成，即并行地处理32位数的4字节。

计算机的发展经历了8位、16位、32位，现在正处于64位时代。然而由位升级带来的性能改善是存在瓶颈的，这也正是短期内我们无法步入128位时代的原因。

指令级（instruction-level）并行

现代CPU的并行度很高，其中使用的技术包括流水线、乱序执行和猜测执行等。

程序员通常可以不关心处理器内部并行的细节，因为尽管处理器内部的并行度很高，但是经过精心设计，从外部看上去所有处理都像是串行的。

而这种“看上去像串行”的设计逐渐变得不适用。处理器的设计者们为单核提升速度变得越来越困难。进入多核时代，我们必须面对的情况是：无论是表面上还是实质上，指令都不再串行执行了。我们将在2.2节的“内存可见性”部分展开讨论。

数据级（data）并行

数据级并行（也称为“单指令多数据”，SIMD）架构，可以并行地在大量数据上施加同一操作。这并不适合解决所有问题，但在适合的场景却可以大展身手。

图像处理就是一种适合进行数据级并行的场景。比如，为了增加图片亮度就需要增加每一个像素的亮度。现代GPU（图形处理器）也因图像处理的特点而演化成了极其强大的数据并行处理器。

任务级（task-level）并行

终于来到了大家所认为的并行形式——多处理器。从程序员的角度来看，多处理器架构最明显的分类特征是其内存模型（共享内存模型或分布式内存模型）。

对于共享内存的多处理器系统，每个处理器都能访问整个内存，处理器之间的通信主要通过内存进行，如图1-1所示。

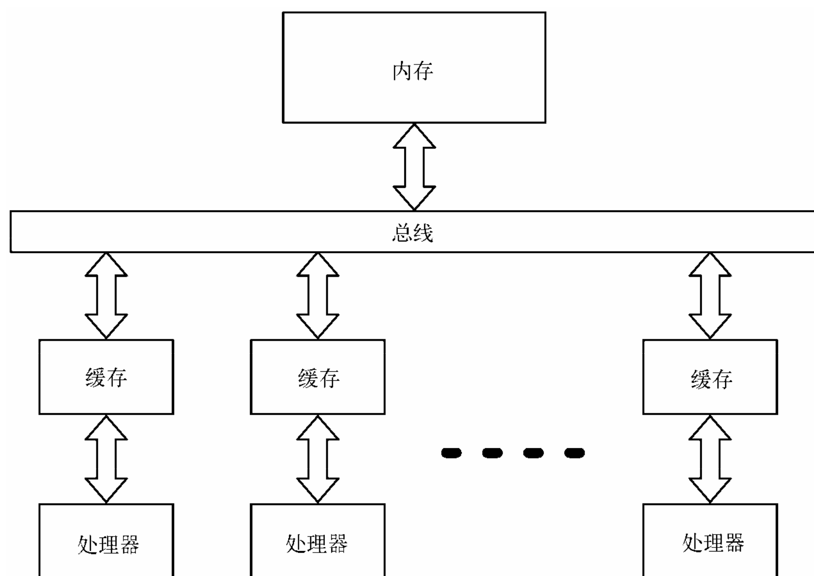


图1-1　共享内存的多处理器系统

对于分布式内存的多处理器系统，每个处理器都有自己的内存，处理器之间的通信主要通过网络进行，如图1-2所示。

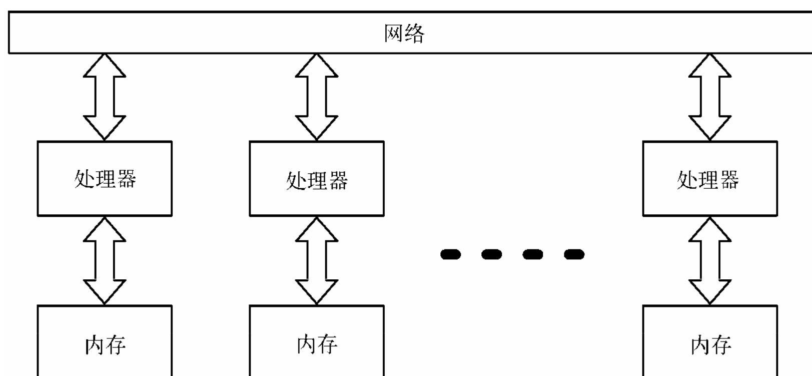


图1-2　分布式内存的多处理器系统

通过内存通信比通过网络通信更简单更快速，所以用共享内存编程往往更容易。然而，当处理器个数逐渐增多，共享内存就会遭遇性能瓶颈——此时不得不转向分布式内存。如果要开发一个容错系统，就要使用多台计算机以规避硬件故障对系统的影响，此时也必须借助于分布式内存。

1.3　并发：不只是多核

使用并发的目的，不仅仅是为了让程序并行运行从而发挥多核的优势。若正确使用并发，程序还将获得以下优点：及时响应、高效、容错、简单。

并发的世界，并发的软件

世界是并发的，为了与其有效地交互，软件也应是并发的。

手机可以同时播放音乐、上网浏览、响应触屏动作。我们在IDE中输入代码时，IDE正在后台悄悄检查代码语法。飞机上的系统也同时兼顾了好几件事情：监控传感器、在仪表盘上显示信息、执行指令、操纵飞行装置调整飞行姿态。

并发是系统及时响应的关键。比如，当文件下载可以在后台进行时，用户就不必一直盯着鼠标沙漏而烦心了。再比如，Web服务器可以并发地处理多个连接请求，一个慢请求不会影响服务器对其他请求的响应。

分布式的世界，分布式的软件

有时，我们要解决地理分布型问题。软件在非同步运行的多台计算机上分布式地运行，其本质是并发。

此外，分布式软件还具有容错性。我们可以将服务器一半部署在欧洲，另一半部署在美国，这样如果一个区域停电就不会造成软件整体不可用。下面就介绍容错性[①]。

不可预测的世界，容错性强的软件

软件有bug，程序会崩溃。即使存在完美的没有bug的程序，运行程序的硬件也可能出现故障。

为了增强软件的容错性，并发代码的关键是*独立性*和*故障检测*。独立性是指一个故障不会影响到故障任务以外的其他任务。故障检测是指当一个任务失败时（原因可能是任务崩溃、失去响应或硬件故障），需要通知负责故障处理的其他任务来处理。

串行程序的容错性远不如并发程序。

复杂的世界，简单的软件

如果曾经花费数小时纠结在一个难以诊断的多线程bug上，那你可能很难接受这个结论，但

① 作者在此处用到了两个词：fault-tolerant和resilient，中文都译为“容错性”，但两者略有区别。由于这种微小的区别不会影响对本书的理解，因此之后的译文不再区分两者，统一使用“容错性”以方便读者理解。——译者注

在选对编程语言和工具的情况下，比起串行的等价解决方案，一个并发的解决方案会更简洁清晰。

在处理现实世界的并发问题时，这个结论可以得到印证。用串行方案解决一个并发问题往往需要付出额外的代价，而且解决方案会晦涩难懂。如果解决方案有着与问题类似的并发结构，就会简单许多：我们不需要创建一个复杂的线程来处理问题中的多个任务，只需要用多个简单的线程分别处理不同的任务即可。

1.4　七个模型

本书精心挑选了七个模型来介绍并发与并行。

线程与锁：线程与锁模型有很多众所周知的不足，但仍是其他模型的技术基础，也是很多并发软件开发的首选。

函数式编程：函数式编程日渐重要的原因之一，是其对并发编程和并行编程提供了良好的支持。函数式编程消除了可变状态，所以从根本上是线程安全的，而且易于并行执行。

Clojure之道——分离标识与状态：编程语言Clojure是一种指令式编程和函数式编程的混搭方案，在两种编程方式上取得了微妙的平衡来发挥两者的优势。

actor：actor模型是一种适用性很广的并发编程模型，适用于共享内存模型和分布式内存模型，也适合解决地理分布型问题，能提供强大的容错性。

通信顺序进程（Communicating Sequential Processes，CSP）：表面上看，CSP模型与actor模型很相似，两者都基于消息传递。不过CSP模型侧重于传递信息的通道，而actor模型侧重于通道两端的实体，使用CSP模型的代码会带有明显不同的风格。

数据级并行：每个笔记本电脑里都藏着一台超级计算机——GPU。GPU利用了数据级并行，不仅可以快速进行图像处理，也可以用于更广阔的领域。如果要进行有限元分析、流体力学计算或其他的大量数字计算，GPU的性能将是不二选择。

Lambda架构：大数据时代的到来离不开并行——现在我们只需要增加计算资源，就能具有处理TB级数据的能力。Lambda架构综合了MapReduce和流式处理的特点，是一种可以处理多种大数据问题的架构。

以上每种模型都有各自的甜区[①]。请带着以下的问题来阅读之后的章节。

- 这个模型适用于解决并发问题、并行问题，还是两者皆可？
- 这个模型适用于哪种并行架构？
- 这个模型是否有利于我们写出容错性强的代码，或用于解决分布式问题的代码？

下一章将介绍第一个模型：线程与锁模型。

① 球类运动中球拍上最适合击球的区域。——译者注

第2章 线程与锁

线程与锁模型就像一辆福特T型车：驾驶它可以到达目的地，但与新的技术相比，它显得原始且难以驾驭，不可靠还有点儿危险。

抛开那些众所周知的缺点，线程与锁模型仍是开发并发软件的首选技术，它也支撑了本书将要介绍的其他技术。你可能不会直接用到这个模型，但也应该了解它是如何工作的。

2.1 简单粗暴

线程与锁模型其实是对底层硬件运行过程的形式化。这种形式化既是该模型最大的优点，也是它最大的缺点。

线程与锁模型非常简单直接，几乎所有编程语言都以某种形式对其提供了支持，且不对其使用方式加以限制。换句话说，对于不精通该模型的程序员，编程语言没有提供足够的帮助，使得程序容易出错且难以维护。

我们将借助Java语言来学习线程与锁模型，但所述内容也适用于其他语言。第一天，将学习Java的多线程代码、潜在的坑以及一些避免踩坑的原则。第二天，将进一步学习`java.util.concurrent`包提供的工具。第三天，学习一些由标准库提供的并发数据结构，尝试使用其解决一个现实问题。

最佳实践

第一天的学习将从Java提供的底层服务开始。现在的优秀代码很少直接使用底层服务，而是使用将在随后讨论的高层服务。要理解高层服务，我们必须先理解基础的底层服务，但请注意：不应在产品代码上直接使用`Thread`类等底层服务。

2.2　第一天：互斥和内存模型

如果你曾经接触过并发编程，那一定熟悉*互斥*这个概念——用锁保证某一时间仅有一个线程可以访问数据。你也肯定熟悉互斥带来的麻烦，比如说*竞态条件*和*死锁*（如果对此不熟悉也无妨，稍后都会介绍）。

我们会详细讨论实践中使用共享内存带来的一些问题，但首先需要关注更基础更重要的内容——内存模型。如果你认为竞态条件和死锁会导致一系列很奇怪的现象，那就对共享内存的诡异程度拭目以待吧。

想超越自我？让我们从创建一个线程开始。

创建线程

Java中，并发的基本单元是*线程*，可以将线程看作*控制流*（thread of control）。线程之间通过共享内存进行通信。

俗话说：一切编程皆始于“Hello, World!”。我们也不免俗地来个多线程版本：

ThreadsLocks/HelloWorld/src/main/java/com/paulbutcher/HelloWorld.java

```java
public class HelloWorld {

  public static void main(String[] args) throws InterruptedException {
    Thread myThread = new Thread() {
        public void run() {
          System.out.println("Hello from new thread");
        }
      };
    myThread.start();
    Thread.yield();
    System.out.println("Hello from main thread");
    myThread.join();
  }
}
```

这段代码创建并启动了一个`Thread`实例。首先从`start()`开始，`myThread.run()`函数与`main()`函数的余下部分一起并发执行。最后`main`线程调用`join()`来等待`myThread`线程结束（即`run()`函数返回）。

运行这段代码的结果可能是

```
Hello from main thread
Hello from new thread
```

也可能是

```
Hello from new thread
Hello from main thread
```

究竟是哪种运行结果完全取决于哪个线程先执行`println()`（我的测试结果是各占50%）。多线程编程很难的原因之一就是运行结果可能依赖于时序，多次运行的结果并不稳定。



小乔爱问：

Thread.yield的作用？

在多线程版本的“Hello, World!”中我们使用了`Thread.yield()`，根据相关的Java文档，其作用是：

> 通知调度器：当前线程想要让出对处理器的占用。

如果不调用`Thread.yield()`，由于创建新线程要花费一些时间，那么`main`线程几乎肯定会先执行`println()`（当然并不保证一定会如此——稍后我们将学到一个规律：并发编程中如果某事可能会发生，那么不论多艰难它一定会发生，而且可能发生在最不利的时刻）。

试将`Thread.yield()`注释掉，看看会发生什么。如果换成`Thread.sleep(1)`呢？

第一把锁

多个线程同时使用共享内存时，它们往往会“打成一片”。为避免如此，我们可以使用锁达到线程互斥的目的，即某一时间至多有一个线程能持有锁。

先创建两个线程，并使其交互：

ThreadsLocks/Counting/src/main/java/com/paulbutcher/Counting.java

```java
public class Counting {
  public static void main(String[] args) throws InterruptedException {
    class Counter {
      private int count = 0;
      public void increment() { ++count; }
      public int getCount() { return count; }
    }
    final Counter counter = new Counter();
    class CountingThread extends Thread {
      public void run() {
        for(int x = 0; x < 10000; ++x)
          counter.increment();
      }
    }
    CountingThread t1 = new CountingThread();
    CountingThread t2 = new CountingThread();
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println(counter.getCount());
  }
}
```

这段代码创建了一个简单的counter类和两个线程，每个线程都调用counter.increment() 10 000次。这段代码看上去很简单但很脆弱。

运行这段代码，每次都将获得不同的结果。最后三次测试的结果是13850、11867和12616。产生这个结果的原因是两个线程使用counter.count对象时发生了竞态条件（即代码行为取决于各操作的时序）。

如果不能理解上面这段话，那让我们来考虑一下Java编译器是如何解释++count的。其字节码是：

```
getfield #2
iconst_1
iadd
putfield #2
```

即使你不熟悉JVM字节码，也可以揣测出这段代码的意图：getfield #2用于获取count的值，iconst_1和iadd将获得的值加1，putfield #2将更新的值写回count中。这就是通称的读–改–写（read-modify-write）模式。

假如两个线程同时调用increment()，线程1执行getfield #2，获得值42。在线程1执行其他动作之前，线程2也执行了getfield #2，获得值42。糟糕的是，现在两个线程都将获得的值加1，将43写回count中。结果count只被递增了一次，而不是两次。

竞态条件的解决方案是对count进行同步（synchronize）访问。一种方法是使用Java对象原生的内置锁（也被称为互斥锁（mutex）、管程（monitor）或临界区（critical section））来同步对increment()的调用：

ThreadsLocks/CountingFixed/src/main/java/com/paulbutcher/Counting.java

```
class Counter {
  private int count = 0;
➤ public synchronized void increment() { ++count; }
  public int getCount() { return count; }
}
```

线程进入increment()函数时，将获取Counter对象级别的锁，函数返回时将释放该锁。某一时间至多有一个线程可以执行函数体，其他线程调用函数时将被阻塞直到锁被释放（稍后我们将了解到：对于这种只涉及一个变量的互斥场景，使用java.util.concurrnet.atomic包是更好的选择）。

毋庸置疑，对于增加了同步功能的代码，每次执行都将得到正确结果20000。

但前路漫漫——代码中仍隐藏了一个bug，我们马上介绍其中的关窍。

诡异的内存

我们用一个小测试来开场，请猜测一下这段代码的输出：

ThreadsLocks/Puzzle/src/main/java/com/paulbutcher/Puzzle.java

```
Line 1 public class Puzzle {
     -   static boolean answerReady = false;
     -   static int answer = 0;
     -   static Thread t1 = new Thread() {
     5       public void run() {
     -         answer = 42;
     -         answerReady = true;
     -       }
     -     };
    10   static Thread t2 = new Thread() {
     -       public void run() {
     -         if (answerReady)
     -           System.out.println("The meaning of life is: " + answer);
     -         else
    15           System.out.println("I don't know the answer");
     -       }
     -     };
     -
     -   public static void main(String[] args) throws InterruptedException {
    20     t1.start(); t2.start();
     -     t1.join(); t2.join();
     -   }
     - }
```

如果你的第一反应是“竞态条件”，那么恭喜你答对了。根据线程执行的时序，这段代码的输出可能是*The meaning of life is XX*或者*I don't know the answer*。但不止于此，还有一种结果可能是：

```
The meaning of life is: 0
```

什么?！当`answerReady`为`true`时`answer`可能为0吗？这仿佛是第6行和第7行在我们眼皮底下颠倒了执行顺序。

但是乱序执行是完全有可能发生的。以下所述均为事实：

- 编译器的静态优化可以打乱代码的执行顺序；
- JVM的动态优化也会打乱代码的执行顺序；
- 硬件可以通过乱序执行来优化其性能。

比乱序执行更糟糕的是，有时一个线程产生的修改可能对另一个线程不可见。如果将`run()`写成：

```
public void run() {
  while (!answerReady)
    Thread.sleep(100);
  System.out.println("The meaning of life is: " + answer);
}
```

`answerReady`可能不会变成`true`，代码运行后无法退出。

从直觉上来说，编译器、JVM、硬件都不应插手修改原本的代码逻辑。但是，近几年的运行效率提升，尤其是共享内存架构的运行效率提升，都仰仗于此类代码优化。因此我们也无法摆脱此类优化的副作用的影响。

显然，需要有标准来明确告诉我们，可能发生怎样的副作用，这就是Java内存模型。

内存可见性

Java内存模型定义了何时一个线程对内存的修改对另一个线程可见[①]。基本原则是，如果读线程和写线程不进行同步，就不能保证可见性。

我们已经见过一种同步的方法——就是通过获取对象的内置锁。其他的方法包括：开启一个线程并通过`join()`检查线程是否已经终止；使用`java.util.concurrent`包提供的工具。

很容易被忽略的一个重点是：两个线程都需要进行同步。只在其中一个线程进行同步是不够的，这正是之前竞态条件解决方案中潜藏bug的原因：除了`increment()`之外，`getCount()`方法也需要进行同步。否则，调用`getCount()`的线程可能获得一个失效的值（对于前面交互的两个线程，`getCount()`在`join()`之后被调用，因此是线程安全的。但这种设计为其他调用`getCounter()`的代码埋下了隐患）。

至此，我们讨论了竞态条件和内存可见性，这两类问题都可能让多线程程序运行结果出错。下面我们将介绍第三类问题：死锁。

多把锁

综上所述，很容易得出一个结论：让多线程代码安全运行的方法只能是让所有的方法都同步。然而，这也会带来问题。

首先这样做效率低下。如果每个方法都同步，大多数线程会频繁阻塞，使程序失去了并发的意义。问题不止于此，当使用多把锁时（Java中每一个对象都有自己的内置锁[②]），线程之间可能发生死锁。

我们将借助一个学术论文中经常使用的经典模型来诠释死锁——哲学家进餐问题。问题场景是五位哲学家围绕一个圆桌就坐，如图2-1所示，桌上摆着五支（不是五双）筷子。

哲学家的状态可能是“思考”或者“饥饿”。如果饥饿，哲学家将拿起他两边的筷子并进餐一段时间（当然，这些哲学家都是男性，女性在餐桌上会更优雅一些）。进餐结束，哲学家就会放回筷子。

---

① http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4
② 对不同对象的方法进行同步就会用到多把锁。——译者注

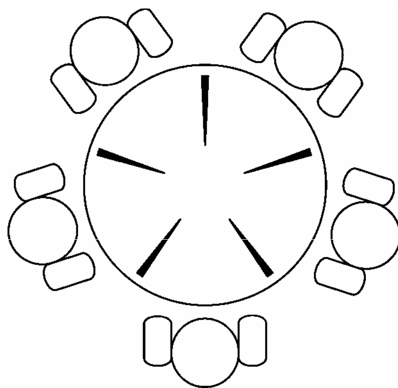


图2-1　哲学家进餐问题

下面的代码实现了哲学家的行为：

ThreadsLocks/DiningPhilosophers/src/main/java/com/paulbutcher/Philosopher.java

```
Line 1 class Philosopher extends Thread {
     -   private Chopstick left, right;
     -   private Random random;
     -
     5   public Philosopher(Chopstick left, Chopstick right) {
     -     this.left = left; this.right = right;
     -     random = new Random();
     -   }
     -
    10   public void run() {
     -     try {
     -       while(true) {
     -         Thread.sleep(random.nextInt(1000));     // 思考一段时间
     -         synchronized(left) {                    // 拿起筷子1
    15           synchronized(right) {                 // 拿起筷子2
     -             Thread.sleep(random.nextInt(1000)); // 进餐一段时间
     -           }
     -         }
     -       }
    20     } catch(InterruptedException e) {}
     -   }
     - }
```

第14、15行使用了另一种方式[①]来获取对象的内置锁：`synchronized(object)`。

在我的计算机上测试过五个哲学家实例，它们可以愉快地运行很久（最长时间是一周），直到某个时刻一切突然都停了下来。

稍加分析就知道发生了什么：如果所有哲学家同时决定进餐，都拿起左手边的筷子，那么就

① 第一种方式是在函数上使用synchronized关键字。——译者注

无法进行下去——所有人都持有一只筷子并等待右手边的人放下筷子。这时死锁就出现了。

一个线程想使用多把锁时，就需要考虑死锁的可能。幸运的是，有一个简单的规则可以避开死锁——总是按照一个全局的固定的顺序获取多把锁。

其中一种实现如下：

ThreadsLocks/DiningPhilosophersFixed/src/main/java/com/paulbutcher/Philosopher.java

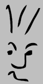
```
class Philosopher extends Thread {
➤ private Chopstick first, second;
  private Random random;

  public Philosopher(Chopstick left, Chopstick right) {
➤   if(left.getId() < right.getId()) {
➤     first = left; second = right;
➤   } else {
➤     first = right; second = left;
➤   }
    random = new Random();
  }

  public void run() {
    try {
      while(true) {
        Thread.sleep(random.nextInt(1000));     // 思考一段时间
➤       synchronized(first) {                   // 拿起筷子1
➤         synchronized(second) {                // 拿起筷子2
            Thread.sleep(random.nextInt(1000)); // 进餐一段时间
          }
        }
      }
    } catch(InterruptedException e) {}
  }
}
```

我们不再按左手边和右手边的顺序拿起筷子，而是按照筷子的编号获得编号1和编号2的锁（我们并不关心编号的具体规则，只要保证编号是全局唯一且有序的）。毫无疑问，现在晚宴将一直愉快地进行下去而不会突然卡住。

不难想到，如果获取锁的代码写得比较集中，就有利于维护这个全局顺序。而对于规模较大的程序，使用锁的地方比较零散，各处都遵守这个顺序就变得不太实际。



小乔爱问：

可以用对象的散列值作为锁的全局顺序吗？

有一个常用的技巧是使用对象的散列值作为锁的全局顺序，类似于下面的代码：

```
if(System.identityHashCode(left) < System.identityHashCode(right)) {
  first = left; second = right;
```

```
    } else {
      first = right; second = left;
    }
```

这个技巧的好处是适用于所有 Java 对象，不用为锁对象专门定义并维护一个顺序。但是对象的散列值并不能保证唯一性（虽然几率很小，但对象的散列值确实可能重复）。我的建议是如果不是迫不得已，不要使用这个技巧。

来自外星方法的危害

规模较大的程序常用监听器模式（listener）来解耦模块。在这里，我们构造一个类从一个URL进行下载，并用`ProgressListeners`监听下载的进度。

ThreadsLocks/HttpDownload/src/main/java/com/paulbutcher/Downloader.java

```java
class Downloader extends Thread {
  private InputStream in;
  private OutputStream out;
  private ArrayList<ProgressListener> listeners;

  public Downloader(URL url, String outputFilename) throws IOException {
    in = url.openConnection().getInputStream();
    out = new FileOutputStream(outputFilename);
    listeners = new ArrayList<ProgressListener>();
  }
  public synchronized void addListener(ProgressListener listener) {
    listeners.add(listener);
  }
  public synchronized void removeListener(ProgressListener listener) {
    listeners.remove(listener);
  }
  private synchronized void updateProgress(int n) {
    for (ProgressListener listener: listeners)
      listener.onProgress(n);
  }

  public void run() {
    int n = 0, total = 0;
    byte[] buffer = new byte[1024];

    try {
      while((n = in.read(buffer)) != -1) {
        out.write(buffer, 0, n);
        total += n;
        updateProgress(total);
      }
      out.flush();
    } catch (IOException e) { }
  }
}
```

addListener()、removeListener()和updateProgress()都是同步方法，多线程可以安全地使用这些方法。尽管这段代码仅使用了一把锁，但仍隐藏着一个死锁陷阱。

陷阱在于updateProgress()调用了一个外星方法——但对于这个外星方法一无所知。外星方法可以做任何事情，例如持有另外一把锁。这样一来，我们就在对加锁顺序一无所知的情况下使用了两把锁。就像前面提到的，这就有可能发生死锁。

唯一的解决思路是避免持有锁时调用外星方法。一种方法是在遍历之前对listeners进行保护性复制（defensive copy），再针对这份副本进行遍历：

ThreadsLocks/HttpDownloadFixed/src/main/java/com/paulbutcher/Downloader.java

```
private void updateProgress(int n) {
  ArrayList<ProgressListener> listenersCopy;
  synchronized(this) {
➤   listenersCopy = (ArrayList<ProgressListener>)listeners.clone();
  }
  for (ProgressListener listener: listenersCopy)
    listener.onProgress(n);
}
```

这是个一石多鸟的方法。不仅在调用外星方法时不用加锁，而且大大减少了代码持有锁的时间。长时间地持有锁将影响性能（降低了程序的并发度），也会增加死锁的可能。保护性复制也修复了与并发无关的另一个bug——修复后如果监听器在onProgress()中调用removeListener()，将不会影响到正在进行遍历的副本。

第一天总结

第一天的学习即将结束。我们通过代码学习了Java多线程的基础，在第二天的学习中将介绍标准库提供的更好的实现方式。

第一天我们学到了什么

本节介绍了如何创建线程，并用Java对象的内置锁实现互斥。还介绍了线程与锁模型带来的三个主要危害——竞态条件、死锁和内存可见性，并讨论了一些帮助我们避免危害的准则：

- 对共享变量的所有访问都需要同步化；
- 读线程和写线程都需要同步化；
- 按照约定的全局顺序来获取多把锁；
- 当持有锁时避免调用外星方法；
- 持有锁的时间应尽可能短。

第一天自习

查找

- 阅读William Pugh的网站“Java内存模型”。

- 自学JSR 133（Java内存模型）的FAQ。
- Java内存模型是如何保证对象初始化是线程安全的？是否必须通过加锁才能在线程之间安全地公开对象？
- 了解反模式“双重检查锁模式”（double-checked locking）以及为什么称其为反模式。

实践

- 对于哲学家进餐问题，用最开始有死锁隐患的代码做一些试验。尝试变更哲学家思考状态的时长、进餐状态的时长以及哲学家的人数。这些变量对于出现死锁的时机有什么影响？设想我们正在进行调试，那应该如何增大重现死锁的几率？
- （困难）编写一段程序，在不使用同步的前提下，模拟内存写操作的乱序执行。这个任务之所以有难度，是因为Java内存模型可能不会优化过于简单的例子，故找到这个优化场景比较困难。

2.3 第二天：超越内置锁

第一天我们学习了Java的Thread类和Java对象的内置锁。在过去的很长一段时间内，这几乎是Java对并发编程提供的所有支持。Java 5通过引入java.util.concurrent包改善了这个状况。今天我们将学习这种增强的锁机制。

内置锁虽然方便但限制很多：

- 一个线程因为等待内置锁而进入阻塞之后，就无法中断该线程了；
- 尝试获取内置锁时，无法设置超时；
- 获得内置锁，必须使用synchronized块。

```
synchronized(object) {
  «使用共享资源»
}
```

这种用法的限制是获取锁和释放锁的代码必须严格嵌在同一个方法中。另外，声明synchronized的函数其实只是个“语法糖”，其等价于将函数体按以下形式进行包装：

```
synchronized(this) {
  «函数体»
}
```

与synchronized不同，ReentrantLock提供了显式的lock和unlock方法，可以突破上述几个限制。

在深入学习之前，先来看一下ReentrantLock是如何替代synchronized工作的：

```
Lock lock = new ReentrantLock();
lock.lock();
try {
```

```
  «使用共享资源»
} finally {
  lock.unlock();
}
```

这段代码中，使用try ... finally是个很好的实践，无论被锁保护的代码发生了什么，都可以确保锁会被释放。

现在我们来看看ReentrantLock是如何突破限制的。

可中断的锁

使用内置锁时，由于阻塞的线程无法被中断，程序不可能从死锁中恢复。我们来看一个小例子，制造一个死锁并尝试中断线程。

ThreadsLocks/Uninterruptible/src/main/java/com/paulbutcher/Uninterruptible.java

```java
public class Uninterruptible {

  public static void main(String[] args) throws InterruptedException {

    final Object o1 = new Object(); final Object o2 = new Object();

    Thread t1 = new Thread() {
      public void run() {
        try {
          synchronized(o1) {
            Thread.sleep(1000);
            synchronized(o2) {}
          }
        } catch (InterruptedException e) { System.out.println("t1 interrupted"); }
      }
    };

    Thread t2 = new Thread() {
      public void run() {
        try {
          synchronized(o2) {
            Thread.sleep(1000);
            synchronized(o1) {}
          }
        } catch (InterruptedException e) { System.out.println("t2 interrupted"); }
      }
    };

    t1.start(); t2.start();
    Thread.sleep(2000);
    t1.interrupt(); t2.interrupt();
    t1.join(); t2.join();
  }
}
```

这段程序将永远死锁下去——跳出死锁唯一的方法是终止JVM的运行。



小乔爱问：

真的没办法终止死锁的线程吗？

你可能认为肯定有某种方法来终止一个死锁线程。遗憾的是确实没有。所有这类方法都被证明有缺陷而不推荐使用。[a]

终止线程的最终手段是让 run()函数返回（可能是通过抛出 InterruptedException）。不过，如果你的线程由于等待内置锁而陷入死锁，且不能中断其等待锁的状态，那么要终止死锁线程就只剩下终止 JVM 运行这条路了。

a. http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

2

不过还是有办法解决这个限制的。我们可以用ReentrantLock替代内置锁，使用它的lockInterruptibly()方法：

ThreadsLocks/Interruptible/src/main/java/com/paulbutcher/Interruptible.java

```
final ReentrantLock l1 = new ReentrantLock();
final ReentrantLock l2 = new ReentrantLock();

Thread t1 = new Thread() {
  public void run() {
    try {
➤     l1.lockInterruptibly();
      Thread.sleep(1000);
➤     l2.lockInterruptibly();
    } catch (InterruptedException e) { System.out.println("t1 interrupted"); }
  }
};
```

这一次Thread.interrupt()可以让线程终止。代码的确比之前稍微复杂一点，这就算是为中断死锁线程付出的一点代价吧。

超时

ReentrantLock突破了内置锁的另一个限制：可以为获取锁的操作设置超时时间。利用这个功能，我们可以通过另一种方法来解决第一天的哲学家进餐问题。

下面是修改后的Philosopher类，拿起两根筷子失败时会超时：

ThreadsLocks/DiningPhilosophersTimeout/src/main/java/com/paulbutcher/Philosopher.java

```
class Philosopher extends Thread {
  private ReentrantLock leftChopstick, rightChopstick;
```

```
    private Random random;

    public Philosopher(ReentrantLock leftChopstick, ReentrantLock rightChopstick) {
      this.leftChopstick = leftChopstick; this.rightChopstick = rightChopstick;
      random = new Random();
    }

    public void run() {
      try {
        while(true) {
          Thread.sleep(random.nextInt(1000)); // 思考一段时间
          leftChopstick.lock();
          try {
➤           if (rightChopstick.tryLock(1000, TimeUnit.MILLISECONDS)) {
              // 获取右手边的筷子
              try {
                Thread.sleep(random.nextInt(1000)); // 进餐一段时间
              } finally { rightChopstick.unlock(); }
            } else {
➤             // 没有获取到右手边的筷子，放弃并继续思考
            }
          } finally { leftChopstick.unlock(); }
        }
      } catch(InterruptedException e) {}
    }
  }
```

这段代码用到了`tryLock()`。相比`lock()`，它在获取锁失败时有超时机制。我们虽然没有遵循“按全局的固定的顺序获取锁”的准则，但这个版本的代码并不会死锁（至少不会无尽地死锁下去）。

活锁

虽然`tryLock()`方案避免了无尽地死锁，但这并不是一个足够好的方案。首先，这个方案并不能避免死锁——它只是提供了从死锁中恢复的手段。其次，这个方案会受到活锁现象的影响——如果所有死锁线程同时超时，它们极有可能再次陷入死锁。虽然死锁没有永远持续下去，但对资源的争夺状况却没有得到任何改善。

有一些方法可以减小活锁的几率。比如为每个线程设置不同的超时时间，来减少所有线程同时超时的几率。但通过设置超时来处理死锁不能说是一个好的方案——以后我们还可以做得更好。

交替锁（hand-over-hand locking）

设想我们要在链表中插入一个节点。一种做法是用锁保护整个链表，但链表加锁时其他使用

者无法访问链表。而交替锁可以只锁住链表的一部分，允许不涉及被锁部分的其他线程自由访问链表，如图2-2所示。

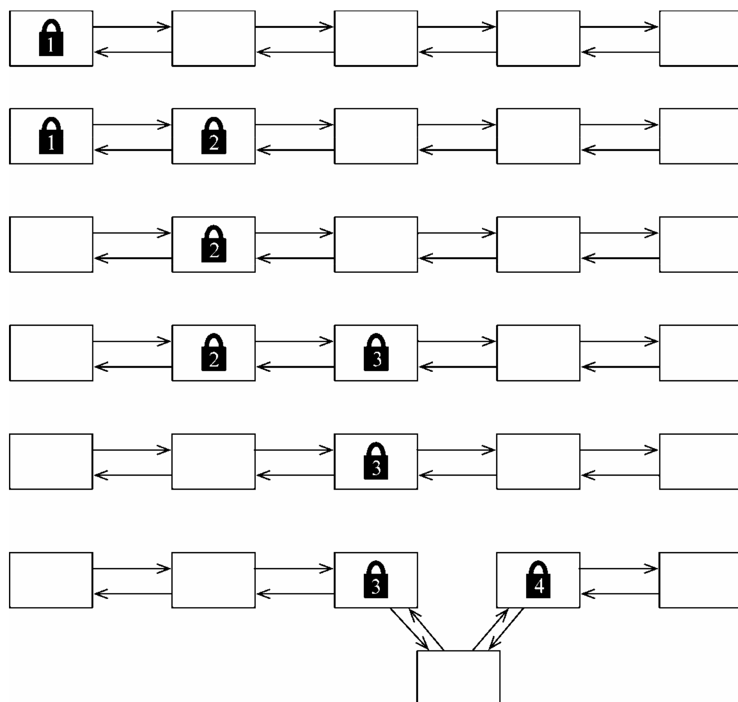


图2-2　交替锁

插入新的链表节点时，需要将待插入位置两边的节点加锁。首先锁住链表的前两个节点。如果这两节点之间不是待插入位置，那么就解锁第一个节点，并锁住第三个节点。如果被锁住的两节点之间仍不是待插入位置，就解锁第二个节点，并锁住第四个节点。以此类推，直到找到待插入位置并插入新的节点，最后解锁两边的节点。

这种交替式的加锁和解锁顺序是无法用内置锁实现的，但使用`ReentrantLock`就可以在需要的地方调用`lock()`和`unlock()`。我们用下面的类来实现使用交替锁的有序链表。

ThreadsLocks/LinkedList/src/main/java/com/paulbutcher/ConcurrentSortedList.java

```
Line 1 class ConcurrentSortedList {
     -
     -   private class Node {
     -     int value;
     5     Node prev;
     -     Node next;
     -     ReentrantLock lock = new ReentrantLock();
     -
     -     Node() {}
```

```
10
-     Node(int value, Node prev, Node next) {
-       this.value = value; this.prev = prev; this.next = next;
-     }
-   }
15
-   private final Node head;
-   private final Node tail;
-
-   public ConcurrentSortedList() {
20    head = new Node(); tail = new Node();
-     head.next = tail; tail.prev = head;
-   }
-
-   public void insert(int value) {
25    Node current = head;
-     current.lock.lock();
-     Node next = current.next;
-     try {
-       while (true) {
30        next.lock.lock();
-         try {
-           if (next == tail || next.value < value) {
-             Node node = new Node(value, current, next);
-             next.prev = node;
35            current.next = node;
-             return;
-           }
-         } finally { current.lock.unlock(); }
-         current = next;
40        next = current.next;
-       }
-     } finally { next.lock.unlock(); }
-   }
- }
```

insert()方法保证链表是有序的：遍历链表直到找到第一个值小于新插入值的节点位置，在这个位置前插入新节点。

第26行锁住了链表的头节点，第30行锁住了下一个节点。接下来检测两个节点之间是否是待插入位置。如果不是，则在第38行解锁当前节点并继续遍历。如果找到待插入位置，第33~36行构造新节点并将其插入链表后返回。两把锁的解锁操作在两个finally块中进行（第38行和第42行）。

这种方案不仅可以让多个线程并发地进行链表插入操作，还能让其他的链表操作安全地并发。比如计算链表节点个数，只需倒序遍历链表即可：

ThreadsLocks/LinkedList/src/main/java/com/paulbutcher/ConcurrentSortedList.java

```
public int size() {
  Node current = tail;
  int count = 0;
```

```
    while (current.prev != head) {
      ReentrantLock lock = current.lock;
      lock.lock();
      try {
        ++count;
        current = current.prev;
      } finally { lock.unlock(); }
    }

    return count;
  }
```




小乔爱问：

难道不会违背“全局顺序”规则吗?

ConcurrentSortedList 的 insert()方法从链表头开始向链表尾依次获取锁。size()方法从链表尾向链表头依次获取锁。难道这样不违背“总是按照一个全局的固定的顺序获取多把锁”的规则吗?

答案是并不违背，因为 size()方法从不持有多把锁——其在某一时间并不持有一把以上的锁。

接下来我们学习 ReentrantLock 的另一个特性——条件变量。

条件变量

并发编程经常需要等待某个事件发生。比如从队列删除元素前需要等待队列非空、向缓存添加数据前需要等待缓存有足够的空间。条件变量就是为这种情况而设计的。

建议按照下面的模式使用条件变量：

```
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
  while (!«条件为真»)
    condition.await();
  «使用共享资源»
} finally { lock.unlock(); }
```

一个条件变量需要与一把锁关联，线程在开始等待条件之前必须获取这把锁。获取锁后，线程检查所等待的条件是否已经为真。如果条件为真，线程将解锁并继续执行。

如果所等待的条件不为真，线程会调用await()，它将原子地解锁并阻塞等待该条件。所谓一个操作是原子的，指的是从另一个线程的角度看上去，该操作的状态只能是“已发生”或者“未发生”，而不会是发生了一半。

当另一个线程调用了signal()或signalAll()，意味着对应的条件可能变为真，await()将原子地恢复运行并重新加锁。需要注意的是当await()函数返回时，只意味着等待的条件可能为真。这就是为什么要在一个循环中调用await()的原因——从await()返回后，需要重新检查等待的条件是否为真，必要的话可能再次调用await()并阻塞。

我们又有了解决“哲学家进餐问题”的新方法：

ThreadsLocks/DiningPhilosophersCondition/src/main/java/com/paulbutcher/Philosopher.java

```
class Philosopher extends Thread {

  private boolean eating;
  private Philosopher left;
  private Philosopher right;
  private ReentrantLock table;
  private Condition condition;
  private Random random;
  public Philosopher(ReentrantLock table) {
    eating = false;
    this.table = table;
    condition = table.newCondition();
    random = new Random();
  }

  public void setLeft(Philosopher left) { this.left = left; }
  public void setRight(Philosopher right) { this.right = right; }

  public void run() {
    try {
      while (true) {
        think();
        eat();
      }
    } catch (InterruptedException e) {}
  }

  private void think() throws InterruptedException {
    table.lock();
    try {
      eating = false;
      left.condition.signal();
      right.condition.signal();
    } finally { table.unlock(); }
    Thread.sleep(1000);
  }

  private void eat() throws InterruptedException {
```

```
      table.lock();
      try {
        while (left.eating || right.eating)
          condition.await();
        eating = true;
      } finally { table.unlock(); }
      Thread.sleep(1000);
    }
  }
```


与之前不同，现在的方法只使用一把锁（table），且没有Chopstick类。我们将竞争从对筷子的争夺转换成了对状态的判断：仅当哲学家的左右邻座都没有进餐时，他才可以进餐。换句话说，一个饥饿的哲学家是在等待下面的条件：

```
!(left.eating || right.eating)
```

当一个哲学家饥饿时，他首先锁住餐桌，这样其他哲学家无法改变状态，然后查看左右邻座是否正在进餐。如果没有，那么该哲学家开始进餐并解锁餐桌。否则其调用await()以解锁餐桌。

当一个哲学家进餐结束并开始思考时，他首先锁住餐桌并将eating设为false，然后通知左右邻座可以进餐了，最后解锁餐桌。如果左右邻座目前正在等待，那么他们将被唤醒，重新锁住餐桌，并判断是否可以开始进餐。

虽然这段代码看上去比之前的解决方案复杂得多，但换来的是并发度的显著提升。在前一个解决方案中，经常出现的状况是只有一个哲学家能进餐，因为其他人都持有一根筷子并在等待另外一根。在这个解决方案中，当一个哲学家理论上可以进餐（他的邻座都没有进餐）时，他肯定可以进餐。

我们已经介绍了ReentrantLock。下面将介绍另一个内置锁的替代方案——原子变量。

原子变量

在第一天的学习中，我们为多线程计数器的increment()方法增加了同步特性（参见2.2节的“第一把锁”部分）。java.util.concurrent.atomic包提供了更好的方案：

ThreadsLocks/CountingBetter/src/main/java/com/paulbutcher/Counting.java

```
public class Counting {
  public static void main(String[] args) throws InterruptedException {

➤   final AtomicInteger counter = new AtomicInteger();

    class CountingThread extends Thread {
      public void run() {
        for(int x = 0; x < 10000; ++x)
➤         counter.incrementAndGet();
      }
    }
```

```
        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();

        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println(counter.get());
    }
}
```

AtomicInteger的incrementAndGet()方法功能上等价于++count（AtomicInteger也提供了getAndIncrement方法，等价于count++）。不过与++count不同，incrementAndGet()方法是原子操作。

与锁相比，使用原子变量有诸多好处。首先，我们不会忘了在正确的时候获取锁。例如，因getCount()忘了同步而引发的Counter内存可见性的问题将不会发生。其次，由于没有锁的参与，对原子变量的操作不会引发死锁。

最后，原子变量是无锁（lock-free）非阻塞（non-blocking）算法的基础，这种算法可以不用锁和阻塞来达到同步的目的。无锁的代码比起有锁的代码更为复杂。幸运的是，java.util.concurrent包中的类都尽量使用了无锁的代码，我们可以在一定程度上免于亲自实现的痛苦。我们将在第三天来学习这些类，而现在先来完成第二天的最后一点内容。



小乔爱问：

volatile变量？

我们可以将Java变量标记成**volatile**。这样可以保证变量的读写不被乱序执行。在之前的测试中（参见2.2节的“诡异的内存”部分），我们可以将answerReady标记成volatile，来解决Puzzle类的问题。

volatile是一种低级形式的同步。它并不能解决Counter的问题（参见2.2节的“第一把锁”部分），因为将count标记成volatile并不能保证count++操作是原子的。

随着JVM被不断优化，其提供了一些低开销的锁，volatile变量的适用场景也越来越少。如果你考虑使用volatile，也许应当在java.util.concurrent.atomic包中寻找更合适的工具。

第二天总结

我们在第一天的基础上，学习了java.util.concurrent.locks包和java.util.concurrent.

atomic包提供的更复杂更灵活的工具。学习和理解这些工具很重要，但经过第三天的学习后我们就会发现实际上很少会直接使用锁。

第二天我们学到了什么

ReentrantLock和java.util.concurrent.atomic突破了使用内置锁的限制，利用新的工具我们可以做到：

- 在线程获取锁时中断它；
- 设置线程获取锁的超时时间；
- 按任意顺序获取和释放锁；
- 用条件变量等待某个条件变为真；
- 使用原子变量避免锁的使用。

第二天自习

查找

- ReentrantLock创建时可以设置一个描述公平性的变量。什么是“公平”的锁？何时适合使用公平的锁？使用非公平的锁会怎样？
- 什么是ReentrantReadWriteLock？它与ReentrantLock有什么区别？适用于什么场景？
- 什么是“虚假唤醒”（spurious wakeup）？什么时候会发生虚假唤醒？为什么符合规范的代码不用担心虚假唤醒？
- 什么是AtomicIntegerFieldUpdater？它与AtomicInteger有什么区别？适用于什么场景？

实践

- 在用条件变量解决哲学家进餐问题时，如果将条件变量所在的循环换成if会发生什么？将看到哪些失败的现象？如果将signal()换成signalAll()会发生什么？会引发什么问题？
- 内置锁比ReentrantLock限制更多，与之类似，内置锁也支持一种限制较多的条件变量。用内置锁、wait()、notify()或notifyAll()重新解决哲学家进餐问题。为什么内置锁比ReentrantLock效率更低？
- 重写ConcurrentSortedList，用一把锁代替交替锁。测试两个方案的性能。交替锁是否有更好的性能？什么情况适用于交替锁？什么情况不适用？

2.4　第三天：站在巨人的肩膀上

java.util.concurrent包不仅提供了第二天介绍的比内置锁更好的锁，还提供了一些通用、高效、bug少的并发数据结构和工具。在实际应用中，较之自己生成解决方案，我们应更多地使

用这些久经考验的工具。

创建线程之终极版

第一天我们学习了如何创建线程，但在实际应用中很少会直接创建线程。举个例子，下面是一个非常简单的服务器，回显接收到的数据：

ThreadsLocks/EchoServer/src/main/java/com/paulbutcher/EchoServer.java

```
public class EchoServer {

  public static void main(String[] args) throws IOException {

    class ConnectionHandler implements Runnable {
      InputStream in; OutputStream out;
      ConnectionHandler(Socket socket) throws IOException {
        in = socket.getInputStream();
        out = socket.getOutputStream();
      }

      public void run() {
        try {
          int n;
          byte[] buffer = new byte[1024];
          while((n = in.read(buffer)) != -1) {
            out.write(buffer, 0, n);
            out.flush();
          }
        } catch (IOException e) {}
      }
    }

    ServerSocket server = new ServerSocket(4567);
    while (true) {
➤     Socket socket = server.accept();
➤     Thread handler = new Thread(new ConnectionHandler(socket));
➤     handler.start();
    }
  }
}
```

标记出来的代码行用于接受一个连接请求并创建一个处理线程。这样的设计虽然能正常工作，但存在两个隐患：第一，创建线程的代价虽然很低，但也没低到能直接忽略的程度，而每个连接都花费了这个代价；第二，如果为每个连接都创建一个线程，当请求连接的速度高于处理连接的速度时，系统的线程数也会随之快速增长，服务器将停止服务甚至崩溃。这就给那些想对服务器进行拒绝服务攻击的人提供了可乘之机。

我们可以用线程池来避免这些问题：

ThreadsLocks/EchoServerBetter/src/main/java/com/paulbutcher/EchoServer.java

```
int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;
ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);
while (true) {
  Socket socket = server.accept();
  executor.execute(new ConnectionHandler(socket));
}
```

这段代码创建了一个线程池，线程池的大小设为可用处理器数的2倍。如果同一时间有超过线程池大小的`execute()`请求存在，超出的部分将进行排队直到某线程被释放。现在我们不必再为每个连接都消耗资源来创建线程[①]，而且服务器在面临高负载时也能继续运转（不能保证服务器对所有连接都及时响应，但至少可以响应其中一部分）。

写入时复制

第一天我们曾学习过在并发程序中如何安全地调用监听器，当时在`updateProgress()`中创建了一个保护性复制（参见2.2节中“来自外星方法的危害”部分）。Java标准库提供了更优雅的现成方案——`CopyOnWriteArrayList`：

ThreadsLocks/HttpDownloadBetter/src/main/java/com/paulbutcher/Downloader.java

```
private CopyOnWriteArrayList<ProgressListener> listeners;

public void addListener(ProgressListener listener) {
  listeners.add(listener);
}
public void removeListener(ProgressListener listener) {
  listeners.remove(listener);
}
private void updateProgress(int n) {
  for (ProgressListener listener: listeners)
    listener.onProgress(n);
}
```

顾名思义，`CopyOnWriteArrayList`使用了保护性复制的策略。它并不是在遍历列表前进行复制，而是在列表被修改时进行，已经投入使用的迭代器会使用当时的旧副本。这种方式对许多用例并不适用，但非常适用于我们当下的场景。

首先，使用了`CopyOnWriteArrayList`的代码会变得非常简洁。事实上除了定义`listeners`的部分稍有不同，其他代码与最初的非线程安全的版本没有什么区别。其次，代码将变得更高效，因为我们不必在每次调用`updateProgress()`时都创建副本，而只在`listeners`被更新时创建即可（更新`listeners`的概率相对较低）。

---

① 新的连接会复用连接池中的已有线程，而不必创建新线程。——译者注



小乔爱问:

线程池应该有多大?

影响线程池最优大小的因素有很多，例如硬件的性能、线程任务是 CPU 密集型还是 IO 密集型、是否有其他任务在同时运行。还有很多其他原因也会产生影响。

话虽如此，但也存在经验法则：对于 CPU 密集型的任务，线程池大小应接近于可用核数；对于 IO 密集型的任务，线程池可以设置得更大些。

当然，最佳的方法是建立一个真实环境下的压力测试来衡量性能。

一个完整的程序

到目前为止，我们单独学习了一些工具。剩下的时间我们将解决一个实际的小问题：Wikipedia 上出现频率最高的词是什么？

乍看上去这应当不难——只需要下载XML dump文件[①]，然后写一个程序解析它们并计算词频就可以了。但dump文件差不多有40 GiB，处理起来需要一些时间，我们是否可以借助并行来加速运行？

先从基本场景开始——一个简单的串行程序统计前100 000页（page）的词频需要花费多久？

ThreadsLocks/WordCount/src/main/java/com/paulbutcher/WordCount.java

```
public class WordCount {
  private static final HashMap<String, Integer> counts =
    new HashMap<String, Integer>();

  public static void main(String[] args) throws Exception {
    Iterable<Page> pages = new Pages(100000, "enwiki.xml");
    for(Page page: pages) {
      Iterable<String> words = new Words(page.getText());
      for (String word: words)
        countWord(word);
    }
  }
  private static void countWord(String word) {
    Integer currentCount = counts.get(word);
    if (currentCount == null)
      counts.put(word, 1);
    else
```

① http://dumps.wikimedia.org/enwiki/

```
      counts.put(word, currentCount + 1);
    }
  }
```

在我的MacBook Pro上，这需要花费105秒。

那应该从何处下手研发并行的版本呢？主循环的每一次循环都完成了两个任务——首先解析XML并构造一个Page，然后“消费”这个page，对page中的内容统计词频。

这类问题可以归结为一种经典模式——生产者-消费者（producer-consumer）模式。相比只用一个线程自产自销，我们可以创建两个线程：一个生产者和一个消费者。

首先，定义一个生产者Parser：

ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/Parser.java

```
class Parser implements Runnable {
  private BlockingQueue<Page> queue;

  public Parser(BlockingQueue<Page> queue) {
    this.queue = queue;
  }

  public void run() {
    try {
➤     Iterable<Page> pages = new Pages(100000, "enwiki.xml");
➤     for (Page page: pages)
➤       queue.put(page);
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

run()的方法体是之前串行版本的外层循环，但对所产生的page不直接统计词频，而是将该page加入到队列末尾。

然后，定义一个消费者：

ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/Counter.java

```
class Counter implements Runnable {
  private BlockingQueue<Page> queue;
  private Map<String, Integer> counts;
  public Counter(BlockingQueue<Page> queue,
                 Map<String, Integer> counts) {
    this.queue = queue;
    this.counts = counts;
  }

  public void run() {
    try {
      while(true) {
➤       Page page = queue.take();
        if (page.isPoisonPill())
```

```
            break;

➤         Iterable<String> words = new Words(page.getText());
➤         for (String word: words)
➤           countWord(word);
        }
      } catch (Exception e) { e.printStackTrace(); }
    }
  }
```

你可能已经猜到了，方法体是之前串行版本的内层循环，从队列里获取输入。

最后，创建两个线程：

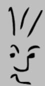
ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/WordCount.java

```
ArrayBlockingQueue<Page> queue = new ArrayBlockingQueue<Page>(100);
HashMap<String, Integer> counts = new HashMap<String, Integer>();

Thread counter = new Thread(new Counter(queue, counts));
Thread parser = new Thread(new Parser(queue));

counter.start();
parser.start();
parser.join();
queue.put(new PoisonPill());
counter.join();
```

`java.util.concurrent`包中的`ArrayBlockingQueue`是一个并发队列，非常适合实现生产者-消费者模式。其提供了高效的并发方法`put()`和`take()`，这些方法会在必要时阻塞：当对一个空队列调用`take()`时，程序会阻塞直到队列变为非空；当对一个满队列调用`put()`时，程序会阻塞直到队列有足够空间。



小乔爱问：

为什么用阻塞队列？

`java.util.concurrent` 包不仅提供了阻塞队列，还提供了一种容量无限、操作不需等待、非阻塞的队列 `ConcurrentLinkedQueue`。这些特性听上去非常诱人，那为什么在这个场景下它不是一个好的解决方案呢？

关键在于生产者和消费者可能不会（几乎肯定不会）保持相同的速度。比如，当生产者的速度快于消费者的速度时，队列会越来越大。Wikipedia 的 dump 差不多有 40 GiB，很容易就让队列大小超过内存容量。

相比之下，阻塞队列只允许生产者的速度在一定程度上超过消费者的速度，但不会超过很多。

另一个有趣的话题是消费者如何知道何时应该退出：

ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/Counter.java

```
if (page.isPoisonPill())
  break;
```

毒丸（poison pill）是一个特殊的对象，告诉消费者“数据已经取完了，你可以退出了”。这非常类似于C/C++中用null字符作为字符串的结尾。

用生产者–消费者模式进行优化后，程序运行提速了——从105秒提升到了95秒。

而我们可以做得更好。生产者–消费者模式的优点不仅在于可以并行地生产和消费，还可以存在多个生产者和/或多个消费者。

那么我们应该重点加速生产者的速度还是消费者的速度？哪段代码占用了大量的运行时间？如果临时改动一下代码，只运行生产者的部分，会发现分析前100 000页花费了近10秒。

仔细想一下就可以解释这个现象。最初的串行版本花费了105秒，而生产者–消费者版本花费了95秒。显然解析文件花费了10秒，而统计词频花费了95秒。所以当解析和统计并行时，整体运行时间会减少到两者中较长的时间——95秒。

要进一步优化，就要对统计过程进行并行化，建立多个消费者。图2-3示意了我们要做的事情。

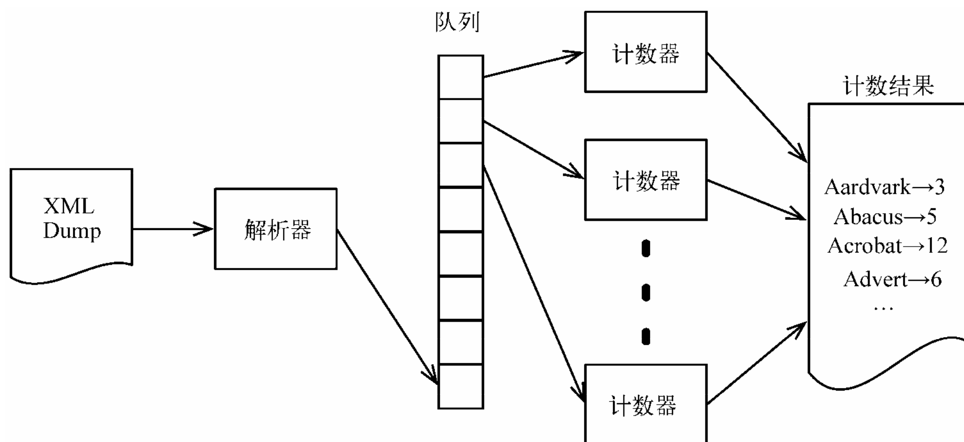


图2-3 建立多个消费者，对统计过程进行并行化

如果多个线程要同时统计词频，就需要一种方法来同步对`counts`对象的访问。

首先，我们想到由`Collections`包的`synchronizedMap()`提供的同步的map。遗憾的是这类同步的集合并不提供原子的读–改–写的方法，所以不能使用它们。如果使用`HashMap`，就必须自己实现对访问的同步。

下面是修改后的countWord()：

ThreadsLocks/WordCountSynchronizedHashMap/src/main/java/com/paulbutcher/Counter.java

```
private void countWord(String word) {
➤ lock.lock();
  try {
    Integer currentCount = counts.get(word);
    if (currentCount == null)
      counts.put(word, 1);
    else
      counts.put(word, currentCount + 1);
➤ } finally { lock.unlock(); }
}
```

然后，修改代码来运行多个消费者：

ThreadsLocks/WordCountSynchronizedHashMap/src/main/java/com/paulbutcher/WordCount.java

```
ArrayBlockingQueue<Page> queue = new ArrayBlockingQueue<Page>(100);
HashMap<String, Integer> counts = new HashMap<String, Integer>();
ExecutorService executor = Executors.newCachedThreadPool();
for (int i = 0; i < NUM_COUNTERS; ++i)
  executor.execute(new Counter(queue, counts));
Thread parser = new Thread(new Parser(queue));
parser.start();
parser.join();
for (int i = 0; i < NUM_COUNTERS; ++i)
  queue.put(new PoisonPill());
executor.shutdown();
executor.awaitTermination(10L, TimeUnit.MINUTES);
```

与之前的主代码相比，这段代码的变化是使用了线程池，方便管理多个线程。我们还必须使用适当数量的毒丸，保证消费者的线程都可以退出。

一切看起来都很完美，但我们的梦想很快就破灭了。分别测量一下使用一个消费者和两个消费者所花费的时间（加速比①是相对于串行版本），如表2-1所示。

表2-1　使用一个消费者和两个消费者所花费时间的比较

消费者	时间（秒）	加速比
1	101	1.04
2	212	0.49

为什么增加一个消费者反而更慢？而且慢了原来的一半之多？

答案是因为过度竞争——过多的线程尝试同时使用一个共享资源。在我们的程序中，消费者花费大量时间等待被其他消费者锁住的counts，它们的等待时间比实际运算时间还要长，最终

① 加速比是指串行算法执行时间和并行算法执行时间的比值。——译者注

导致惨烈的性能下降。

好在我们不会就此退缩。java.util.concurrent包的ConcurrentHashMap正是我们所需要的。它不仅提供了原子的读-改-写方法，还使用了更高级的并发访问（被称为锁分段（lock striping）技术）[①]。

下面是使用了ConcurrentHashMap的countWord()代码。

ThreadsLocks/WordCountConcurrentHashMap/src/main/java/com/paulbutcher/Counter.java

```
private void countWord(String word) {
  while (true) {
    Integer currentCount = counts.get(word);
    if (currentCount == null) {
      if (counts.putIfAbsent(word, 1) == null)
        break;
    } else if (counts.replace(word, currentCount, currentCount + 1)) {
      break;
    }
  }
}
```

我们需要花点时间来理解这个机制。此处使用了putIfAbsent()和replace()来取代原来的put()方法。putIfAbsent()的相关文档如下。

> 如果指定键没有与某值关联，则将指定键与指定值进行关联。其与以下代码的区别是具有原子性：

```
if (!map.containsKey(key))
  return map.put(key, value);
else
  return map.get(key);
```

replace()的相关文档如下。

> 仅当指定键与指定旧值关联时，将指定键与指定新值进行关联。其与以下代码的区别是具有原子性：

```
if (map.containsKey(key) && map.get(key).equals(oldValue)) {
  map.put(key, newValue);
  return true;
} else return false;
```

当使用这些函数时，需要检查其返回值来判断是否操作成功。如果没有成功，则需要重试。

再次测量运行时间，这次就不那么悲剧了，如表2-2所示。

① ConcurrentHashMap内部使用了锁分段技术，可以提升其并发性能。——译者注

表2-2　使用Concurrent HashMap所花费时间的比较

消费者	时间（秒）	加速比
1	120	0.87
2	83	1.26
3	65	1.61
4	63	1.67
5	70	1.50
6	79	1.33

搞定！这次可以通过增加消费者的数量来提升计算速度。不过增加到4个以上的消费者时，计算速度开始下降。

虽然63秒的成绩比串行版本的105秒要好得多，但也没能达到2倍的提速。我的MacBook有四个核——理论上应该有近4倍的提速。

我们还注意到消费者对`counts`有一些不必要的竞争。与其所有消费者都共享同一个`counts`，不如每个消费者各自维护一个计数map，再对这些计数map进行合并：

ThreadsLocks/WordCountBatchConcurrentHashMap/src/main/java/com/paulbutcher/Counter.java

```
private void mergeCounts() {
  for (Map.Entry<String, Integer> e: localCounts.entrySet()) {
    String word = e.getKey();
    Integer count = e.getValue();
    while (true) {
      Integer currentCount = counts.get(word);
      if (currentCount == null) {
        if (counts.putIfAbsent(word, count) == null)
          break;
      } else if (counts.replace(word, currentCount, currentCount + count)) {
        break;
      }
    }
  }
}
```

这次离理想的4倍提速又近了一步，如表2-3所示。

表　2-3

消费者	时间（秒）	加速比
1	95	1.10
2	57	1.83
3	40	2.62
4	39	2.69
5	35	2.96
6	33	3.14
7	41	2.55

现在的程序性能不仅随着消费者的增加而快速提升，而且超过4个消费者后性能仍会继续提升。这大概是因为我的MacBook支持“超线程”——虽然只有4个物理核，但是`availableProcessors()`会返回8。

图2-4展示了三个不同版本程序的性能。

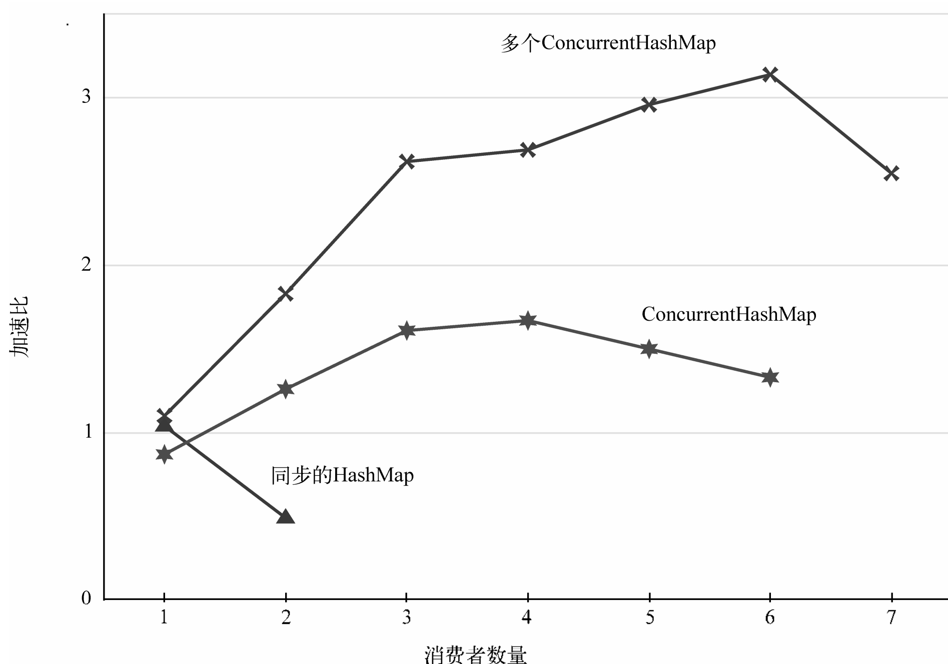


图2-4 消费者个数对词频统计程序性能的影响

并行程序的性能曲线大多与此类似。起初性能快速线性增长，之后增长趋势放缓，最终性能达到极值，线程数再增加性能则会下降。

现在回顾一下我们已经完成了什么：建立了一个相对复杂的生产者–消费者程序，多个消费者通过一个并发队列和一个并发map进行协作，程序中没有显式地使用锁，而是使用了标准库提供的并发工具。

第三天总结

我们已经完成了线程与锁模型最后一天的学习。

第三天我们学到了什么

`java.util.concurrent`包提供的工具不仅让并发编程更容易，而且在以下方面让程序更安

全高效：

- 使用线程池，而不直接创建线程；
- 使用`CopyOnWriteArrayList`让监听器相关的代码更简单高效；
- 使用`ArrayBlockingQueue`让生产者和消费者之间高效协作；
- `ConcurrentHashMap`提供了更好的并发访问。

第三天自习

查找

- 阅读`ForkJoinPool`的文档——fork/join框架与线程池有什么区别？分别适用于什么场景？
- 什么是work-stealing？它适用于什么场景？如何用`java.util.concurrent`包提供的工具实现work-stealing？
- `CountDownLatch`和`CyclicBarrier`有什么区别？分别适用于什么场景？
- 什么是阿姆达尔定律（Amdahl’s law）？如何计算出词频统计程序的最大理论加速比？

实践

- 修改生产者–消费者版本的代码，用“数据结束”标识取代毒丸，在生产者与消费者有速度差异时要确保程序运行正常。队列被置“数据结束”标识时，如果消费者尝试从队列中移除元素会发生什么？为什么毒丸方法会被广泛采用？
- 在不同的电脑上运行不同版本的词频统计程序。不同电脑上的性能曲线有什么区别？如果在一台32核电脑上运行，是否会得到近32倍的提速？

2.5 复习

线程与锁模型可能是这本书介绍的最具有争议的模型。很多程序员觉得它难以驾驭而感到惧怕，不顾一切地避免多线程编程。而另一些程序员却不以为然，他们觉得只要遵守一些规则，线程与锁模型其实别无他样。

让我们来看一下这个模型的优点和缺点。

优点

线程与锁模型的最大优点是其适用面很广。它是本书介绍的其他许多技术的基础，适用于解决很多类型的问题。同时，线程与锁模型更接近于“本质”——近似于对硬件工作方式的形式化——正确使用时，其效率很高。这也意味着它能够解决从小到大不同粒度的问题。

另外，这个模型可以被轻松地集成到大多数编程语言中。语言设计者们可以轻易让一门指令式语言或面向对象语言支持线程与锁模型。

缺点

线程与锁模型没有为并行提供直接的支持（之前我们提到过并发与并行的区别和联系，参见1.1节）。在之前词频统计的例子中，我们确实用这个模型将一个串行算法并行地运行，但前提是该程序被改造成了并发形式，同时引入了不确定性的隐患。

2

除了一些特例，比如实验性的研究分布式共享内存的系统，线程与锁模型仅支持共享内存模型。如果要支持分布式内存模型（无论是地理分布型或者容错型），就需要寻求其他技术的帮助。这也意味着线程与锁模型不适用于单个系统无力解决的问题。

使用这个模型最大的缺点在于*无助*。语言设计者很容易将其集成到某一门语言中，但对于我们这些可怜的程序员，编程语言层面并没有提供足够的帮助。

不易察觉的错误

我认为应用多线程的难点不在于难以编程，而在于难以*测试*。在多线程编程中，从坑里爬出来并不难，难的是我们不知道自己是不是已在坑里。

以内存模型为例。如2.2节的“内存可见性”部分所述，两个线程在没有同步的情况下访问同一片内存，奇怪的事情就会接踵而至。但我们怎么知道自己的程序是不是正确的？是否能写一个测试来证明访问内存时相应的代码都有同步保护？

可惜我们做不到。确实可以写一段代码来进行压力测试，但这些测试成功并不意味着被测代码是正确的。例如，在解决哲学家进餐问题时，我们曾讨论过产生死锁的代码，而这段代码曾经正常运行了一周多后才出现死锁。

测试中的一个大问题是多线程的bug很难重现——我不止一次在凌晨被叫醒，因为服务器在正常运行几个月后突然出了问题。如果一个问题每十分钟就会发生一次，我们很快就能定位该问题，但如果要正常运行几个月才能重现问题，这就很难进行调试。

更糟糕的是，我们很有可能写出一个包含多线程bug的程序，但无论怎样彻底地长期地进行测试，该程序从不会被测出错误来。假如用一种可能被乱序执行的方式访问内存，并不意味着乱序执行真的会发生。这样，我们就完全不知道程序有bug，直到升级JVM或者使用不同的硬件时，才会发生一些诡异的事情。

可维护性

上述问题在编写代码时已经让人很头疼了，更过分的是代码不可能不变更。我们要全程保证所有对象的同步都是正确的、必须按照顺序来获取多把锁、持有锁时不调用外星方法。还要保证12个月之后换了另外10个程序员仍然按照这个规则维护代码。过去十几年，自动测试让我们信心十足地进行重构，但如果不能对多线程问题进行可靠的测试，就无法对多线程代码进行可靠的重构。

最终我们仅有的方法是谨慎地思考多线程代码。除了谨慎地思考，就是更谨慎地思考。当然，这种方法并不容易，而且无法量化。

收拾残局

我认为诊断多线程问题的感觉，非常类似于一级方程式赛车的工程师诊断引擎故障：引擎在正常运行几个小时后，突然在没有任何征兆的情况下发生严重故障，机油和零件散落一地，狼狈不堪。

当赛车被拖回维修厂后，可怜的工程师要面对一堆残骸找出故障的原因。故障原因可能是很小的——一个坏的油泵轴承或者阀门，但应该如何从一片混乱中找出原因呢？

经常使用的方法是尽可能地完善对引擎数据的记录，并让赛车手使用新的引擎。希望下次发生故障时数据能提供一些有用的信息。

其他语言

如果你想深入研究JVM的线程与锁模型，由`java.util.concurrent`包的作者撰写的*Java Concurrency in Practice* [Goe06]是个不错的起点。不同语言之间，多线程编程的细节可能有所不同，但本章我们介绍的原理普遍适用，包括下面这些规则：访问共享变量时需要同步、按照全局的固定的顺序来获得多把锁、持有锁时避免调用外星方法。

值得一提的是，虽然我们仅讨论了Java的内存模型，但是会对内存访问进行乱序执行的却不止Java。大多数语言没有对内存模型做出完善的定义，没有明确地说明乱序执行何时发生以及如何发生。在这方面Java是先驱者，是第一个完整定义内存模型的主流语言。C和C++是在C 11和C++ 11的标准中才补充了内存模型。

结语

伴随着种种挑战，在可预见的将来多线程编程将一直伴随着我们。我们也会在之后的章节中介绍其他一些有用的相关知识。

下一章我们将学习函数式编程，它不使用可变状态，从而避免了线程与锁模型的很多缺陷。即使你不打算写函数式的代码，理解函数式编程背后的原理也很有价值——我们以后学习的很多并发模型也应用了这些原理。

第3章

函数式编程

函数式编程（Functional Programming）就像一辆高端、新潮的氢燃料汽车，虽然还未被广泛使用，但二十年后我们的生活将与它密不可分。

函数式编程与命令式编程（Imperative Programming）不同。命令式编程的代码由一系列改变全局状态的语句构成，而*函数式编程*则是将计算过程抽象成表达式求值。这些表达式由纯数学函数构成，而这些数学函数是第一类对象（我们可以像操作数值一样操作第一类对象）并且没有副作用。由于没有副作用，函数式编程可以更容易做到线程安全，因此特别适合于并发编程。这也是我们学习的第一个可以直接支持并行的模型。

3.1 若不爽，就另辟蹊径

第2章提到的有关锁的一些规则，都是针对于线程之间共享的可变的数据——换个说法就是*共享可变状态*。而对于不变的数据，多线程不使用锁就可以安全地进行访问。

这就是为什么在解决并发和并行问题时函数式编程会如此引人注目——它没有可变状态，所以不会遇到由共享可变状态带来的种种问题。

本章中我们将用Clojure语言①来介绍函数式编程，这是一门运行在JVM上的Lisp方言。Clojure是动态类型语言，如果你是Ruby或Python程序员，肯定不会对此感到陌生。虽然Clojure不是一门纯粹的函数式语言，但本章只会讨论其函数式的特征。本书只能按需介绍Clojure，如果你还想深入学习，推荐阅读Stuart Halloway和Aaron Bedra编写的《Clojure程序设计》②。

第一天，我们将学习函数式编程的基础，并了解如何并行化一个函数式算法。第二天，深入学习Clojure的reducer框架，以及在该框架下是如何进行并行化的。第三天，将注意力从并行转向并发，利用future模型和promise模型创建一个并发的函数式Web服务。

① http://clojure.org

② 该书中文版已由人民邮电出版社出版。——编者注

3.2　第一天：抛弃可变状态

当程序员初次学习函数式编程时，第一反应通常是怀疑——正常程序怎么可能不更新变量。经过后面的学习你会发现，这不仅是可能的，而且要比写命令式的代码更简单。

可变状态的风险

今天将重点讨论并行。我们将创建一个简单的函数式程序，并演示如何轻松地将其并行化。

不过要先来学习几个Java的例子，看看为什么要避免使用可变状态。

隐藏的可变状态

下面这个类没有使用可变状态，看上去肯定是线程安全的：

FunctionalProgramming/DateFormatBug/src/main/java/com/paulbutcher/DateParser.java

```
class DateParser {
  private final DateFormat format = new SimpleDateFormat("yyyy-MM-dd");

  public Date parse(String s) throws ParseException {
    return format.parse(s);
  }
}
```

但当多个线程使用这个类的同一对象时（源码可以在本书配套代码中找到），首次运行可能会得到如下错误：

```
Caught: java.lang.NumberFormatException: For input string: ".12012E4.12012E4"
Expected: Sun Jan 01 00:00:00 GMT 2012, got: Wed Apr 15 00:00:00 BST 2015
```

再次运行，可能又会得到如下错误：

```
Caught: java.lang.ArrayIndexOutOfBoundsException: -1
```

第三次运行，可能还会得到另一个错误：

```
Caught: java.lang.NumberFormatException: multiple points
Caught: java.lang.NumberFormatException: multiple points
```

虽然这段代码只有一个成员变量，而且被标记成不可变（即`final`），但显然这段代码根本达不到线程安全，为什么呢？

造成问题的原因是`SimpleDateFormat`内部有隐藏的可变状态。你可能会认为这应该是个bug[①]，但对我们来说是不是bug并不重要。Java这类语言为了让代码写起来简单，在此隐藏了可变状态，这也使我们无法判断何时会发生问题——从API无法判断`SimpleDateFormat`是否是线程安全的。

---

① http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4228335

隐藏的可变状态还不是唯一需要留意的问题，我们再来看一个。

逃逸的可变状态

假设我们要创建一个管理比赛的Web服务。需求是能管理一个运动员列表，我们会习惯地写出如下代码：

```
public class Tournament {
  private List<Player> players = new LinkedList<Player>();

  public synchronized void addPlayer(Player p) {
    players.add(p);
  }

  public synchronized Iterator<Player> getPlayerIterator() {
    return players.iterator();
  }
}
```

乍一看上去这段代码是线程安全的——`players`是私有变量，仅被`addPlayer()`和`getPlayerIterator()`使用，且两个方法都标记了`synchronized`。然而它并不是线程安全的，因为`getPlayerIterator()`返回的迭代器仍引用了`players`内部的可变状态——如果在迭代器被使用时，另一个线程调用了`addPlayer()`方法，那么程序就会抛出`ConcurrentModificationException`或者变得更糟。也就是说可变状态从`Tournament`的重重防护下逃逸了。

在并发编程中，隐藏和逃逸仅仅是两种可变状态带来的风险——还有很多其他风险。如果能找到一种不使用可变状态的方法，就可以避开这些风险，这正是函数式编程为我们带来的曙光。

Clojure旋风之旅

了解Clojure的Lisp语法只需要几分钟。

体验Clojure最简单的方法是使用它的REPL（Read-Evaluate-Print Loop），可以通过`lein repl`来启动REPL（`lein`是Clojure的构建工具）。在REPL中输入代码，代码将被立刻执行，既不需要创建源文件，也不需要编译，这在进行代码试验时会出人意料地方便。REPL启动后，会看到如下提示符：

```
user=>
```

在提示符后输入的Clojure代码将立刻被执行。

Clojure代码由s-表达式构成，可以将s-表达式视为带括号的列表。主流语言中的`max(3, 5)`在Clojure中写成：

```
user=> (max 3 5)
5
```

数学运算符也是同样的表示方式。比如`1 + 2 * 3`，写成：

```
user=> (+ 1 (* 2 3))
7
```

使用def可以定义常量：

```
user=> (def meaning-of-life 42)
#'user/meaning-of-life
user=> meaning-of-life
42
```

控制结构也可以写成s-表达式：

```
user=> (if (< meaning-of-life 0) "negative" "non-negative")
"non-negative"
```

Clojure的大多数语句都是一个s-表达式，然而也有个别例外。矢量（数组）字面量是用方括号表示的：

```
user=> (def droids ["Huey" "Dewey" "Louie"])
#'user/droids
user=> (count droids)
3
user=> (droids 0)
"Huey"
user=> (droids 2)
"Louie"
```

map字面量是用花括号表示的：

```
user=> (def me {:name "Paul" :age 45 :sex :male})
#'user/me
user=> (:age me)
45
```

map的键通常是关键字（keyword）[①]，关键字以冒号开头，非常类似于Ruby中的符号（symbol）或Java中的保留字符串（interned string）。

使用defn可以定义函数，函数参数是矢量形式的：

```
user=> (defn percentage [x p] (* x (/ p 100.0)))
#'user/percentage
user=> (percentage 200 10)
20.0
```

Clojure旋风之旅就此结束。以后我们还会介绍Clojure的其他特性。

第一个函数式程序

我们一直说函数式编程最有趣的地方是不使用可变状态，但一直没有举例说明。现在就来补充一个例子。

① 此处关键字是指Clojure中的数据类型，而并非编程语言中的for、while这一类关键字。——译者注

对一个数列求和，如果使用Java这种命令式语言，会写成下面这样：

```
public int sum(int[] numbers) {
  int accumulator = 0;
  for (int n: numbers)
    accumulator += n;
  return accumulator;
}
```

这段代码中accumulator是可变的：在for循环中每次都会更新值，因此这段代码不是函数式的。相比之下，使用Clojure的方案可以不使用可变状态：

FunctionalProgramming/Sum/src/sum/core.clj

```
(defn recursive-sum [numbers]
  (if (empty? numbers)
    0
    (+ (first numbers) (recursive-sum (rest numbers)))))
```

这是一个*递归的*的方案——recursive-sum递归地调用自己。如果numbers为空，则返回0。否则，将numbers的第一个元素（first）与其他元素（rest）的和相加，并返回结果。

这个方案是可行的，但还能做得更好。下面这种方案更简单高效：

FunctionalProgramming/Sum/src/sum/core.clj

```
(defn reduce-sum [numbers]
  (reduce (fn [acc x] (+ acc x)) 0 numbers))
```

这段代码使用了Clojure的reduce函数，其有3个参数——一个化简函数、一个初始值和一个集合。

代码中用fn定义了一个匿名化简函数，其接受两个参数并返回参数的和。这个匿名化简函数被传给reduce，reduce为集合中的每一个元素都调用一次化简函数——第一次，初始值（本例中是0）和集合中的第一个元素被传给化简函数；第二次，将第一次调用的结果和集合中的第二个元素传给化简函数；以此类推。

先别鼓掌，我们还可以继续改进——+是个现成的函数，接受两个参数并返回参数的和。直接用+来替换匿名化简函数：

FunctionalProgramming/Sum/src/sum/core.clj

```
(defn sum [numbers]
  (reduce + numbers))
```

现在可以鼓掌了。我们得到了一个比命令式编程更简单明了的方案，这种体验可以说是将命令式方案转换成函数式方案时的普遍感受。



小乔爱问：

如果将空集合传给reduce会发生什么？

sum函数的最终版中我们没有向reduce传初始值：

```
(reduce + numbers)
```

这可能会引发你的思考，如果向reduce传入一个空集合会发生什么？答案是一切运行正常，并且reduce会返回0：

```
sum.core=> (sum [])
0
```

那么，reduce如何知道应该返回0（而不是其他值，比如1或nil）？这涉及Clojure中很多操作符都具有的一个特性——操作符知道自己的特征值（identity value）是什么。以+函数为例，其可以接受任意数目的参数，包括0个参数：

```
user=> (+ 1 2)
3
user=> (+ 1 2 3 4)
10
user=> (+ 42)
42
user=> (+)
0
```

当用0个参数调用函数时，函数会返回加法的特征值，即0。

类似地，*返回乘法的特征值，即1。

```
user=> (*)
1
```

如果不向reduce传初始值，那reduce将使用0个参数来调用函数，并用其作为初始值。

顺便一提，由于+可以接受若干个参数，因此我们也可以用apply来实现sum函数。apply可以接受一个函数和一个矢量，调用函数时将这个矢量展开作为函数的参数：

FunctionalProgramming/Sum/src/sum/core.clj

```
(defn apply-sum [numbers]
  (apply + numbers))
```

但是，与使用reduce不同，使用apply的代码不能轻易地并行化。

轻松并行

我们已经有了一些函数式的代码，那怎么让它并行呢？需要如何修改sum函数呢？其实只需要做非常小的改动：

FunctionalProgramming/Sum/src/sum/core.clj

```
(ns sum.core
  (:require [clojure.core.reducers :as r]))

(defn parallel-sum [numbers]
  (r/fold + numbers))
```


唯一的修改是这段代码用clojure.core.reducers包（代码中使用其缩写r）提供的fold函数替换了reduce。

下面是REPL的一组运行结果，展示了性能的提升：

```
sum.core=> (def numbers (into [] (range 0 10000000)))
#'sum.core/numbers
sum.core=> (time (sum numbers))
"Elapsed time: 1099.154 msecs"
49999995000000
sum.core=> (time (sum numbers))
"Elapsed time: 125.349 msecs"
49999995000000
sum.core=> (time (parallel-sum numbers))
"Elapsed time: 236.609 msecs"
49999995000000
sum.core=> (time (parallel-sum numbers))
"Elapsed time: 49.835 msecs"
49999995000000
```

这段代码先用into函数将0到10 000 000之间的数字添加到一个空矢量中，以此构造一个初始矢量。接着使用time来打印某段代码的运行时间。由于代码是运行在JVM中，就需要多次运行代码，以便预热JIT编译器，这样才能得到比较客观的运行时间。

在我的四核的Mac上，使用fold让代码的运行时间从125 ms降到了50 ms，加速比为2.5。第二天中我们将展开学习fold的实现细节，现在先来看看之前Wikipedia词频统计的例子，实现其函数式版本。

Wikipedia词频统计的函数式版本

我们先来写一个Wikipedia词频统计的串行执行版本——明天再来对它进行并行化。程序需要以下三个函数。

- 函数1，接受Wikipedia XML dump，返回dump中的页面序列。
- 函数2，接受一个页面，返回页面上的词序列。

❑ 函数3，接受一个词序列，返回含有词频的map。

我们不会详细介绍前两个函数——毕竟本书的主题是并发，而不是XML或字符串处理（相关细节请参见本书的配套代码）。在此着重讨论如何统计词频，因为之后我们还将对这部分进行并行化处理。

函数式map

要得到一个包含词频的map，就需要理解Clojure的两个map处理函数——`get`和`assoc`：

```
user=> (def counts {"apple" 2 "orange" 1})
#'user/counts
user=> (get counts "apple" 0)
2
user=> (get counts "banana" 0)
0
user=> (assoc counts "banana" 1)
{"banana" 1, "orange" 1, "apple" 2}
user=> (assoc counts "apple" 3)
{"orange" 1, "apple" 3}
```

`get`从map中查找键，如果找到则返回对应值，否则返回默认值。`assoc`接受一个map和一个键值对，在原有map的基础上返回一个包含指定键值对的新map。

词频统计函数

我们已经准备好实现词频统计函数了，这个函数接受一个词序列，并返回一个map，这个map的键是词，值是该词出现的次数：

FunctionalProgramming/WordCount/src/wordcount/word_frequencies.clj

```
(defn word-frequencies [words]
  (reduce
  (fn [counts word] (assoc counts word (inc (get counts word 0))))
  {} words))
```

这段代码将一个空的map`{}`作为初始值传给`reduce`。再对`words`中的每个元素，将其现有次数加1。试用一下：

```
user=> (word-frequencies ["one" "potato" "two" "potato" "three" "potato" "four"])
{"four" 1, "three" 1, "two" 1, "potato" 3, "one" 1}
```

Clojure标准库已经走在了我们前面——提供了`frequencies`函数，该函数能针对任何集合，输出集合中每个元素的出现次数：

```
user=> (frequencies ["one" "potato" "two" "potato" "three" "potato" "four"])
{"one" 1, "potato" 3, "two" 1, "three" 1, "four" 1}
```

我们已经学会了如何统计词频，剩下的就是将其与XML处理结合起来。

插播一些与序列相关的函数

为了理解今后的代码，得插播一些与序列相关的机制。首先，介绍映射函数map：

```
user=> (map inc [0 1 2 3 4 5])
(1 2 3 4 5 6)
user=> (map (fn [x] (* 2 x)) [0 1 2 3 4 5])
(0 2 4 6 8 10)
```

map函数接受一个函数f和一个序列，并返回一个新序列，对于输入序列中的每个元素都会调用一次函数f，并以元素的值作为f的参数，f的返回值则成为新序列的对应元素。

然后，介绍partial函数，利用partial函数可以将上面代码的第二个调用简化一下。partial接受一个函数和若干参数，返回一个被局部代入的函数[①]：

```
user=> (def multiply-by-2 (partial * 2))
#'user/multiply-by-2
user=> (multiply-by-2 3)
6
user=> (map (partial * 2) [0 1 2 3 4 5])
(0 2 4 6 8 10)
```

最后，假设有一个函数会返回一个序列，比如用正则表达式将字符串切割成词的序列：

```
user=> (defn get-words [text] (re-seq #"\w+" text))
#'user/get-words
user=> (get-words "one two three four")
("one" "two" "three" "four")
```

显然，对一个字符串序列用get-words进行映射，会得到一个二维序列：

```
user=> (map get-words ["one two three" "four five six" "seven eight nine"])
(("one" "two" "three") ("four" "five" "six") ("seven" "eight" "nine"))
```

如果需要包含所有输出的一维序列，则可以使用mapcat：

```
user=> (mapcat get-words ["one two three" "four five six" "seven eight nine"])
("one" "two" "three" "four" "five" "six" "seven" "eight" "nine")
```

一切准备就绪，现在可以组装我们的词频统计函数了。

组装

我们将这个词频统计函数命名为count-words-sequential，它接受一个页面序列，同时返回含有对应词频的map：

FunctionalProgramming/WordCount/src/wordcount/core.clj

```
(defn count-words-sequential [pages]
  (frequencies (mapcat get-words pages)))
```

---

① 举例说明，假设有一个数学函数f(a,b,c)，partial(f, 1)返回的是数学函数f(1,b,c)，函数的参数a已经被代入了。——译者注

这段代码首先用(mapcat get-words pages)将页面序列转化成词序列，再将词序列传给frequencies。

与之前命令式版本（参见2.4节中“一个完整的程序”部分）相比，函数式版本的简洁优美又一次得到印证。

懒惰一点好

你可能有一个困惑——Wikipedia dump的大小将近40 GiB。如果count-words将其中的词都存放到一个序列中，内存应该会不够用。

实际情况并不是这样，因为Clojure中序列是懒惰的（lazy）——其中的元素仅在需要时被求值。举例说明：

range函数会产生一个数列：

```
user=> (range 0 10)
(0 1 2 3 4 5 6 7 8 9)
```

上面的数列在REPL中会被完全求值并输出。

我并不能阻止你对一个超大范围进行求值，但这样做，你的电脑很有可能变成一台高热的暖风机。比如下面的这个例子，需要花费很长时间才会得到输出（假设内存足够用）：

```
user=> (range 0 100000000)
```

再试试下面的例子，能立刻得到输出：

```
user=> (take 10 (range 0 100000000))
(0 1 2 3 4 5 6 7 8 9)
```

由于take只需要数列的前10个元素，因此range只需要产生10个元素。这个机制适用于任意层次的嵌套：

```
user=> (take 10 (map (partial * 2) (range 0 100000000)))
(0 2 4 6 8 10 12 14 16 18)
```

甚至可以使用无穷序列。比如，iterate函数会不断将某个函数应用到初始值、第一次的返回值、第二次的返回值……来构成无穷序列：

```
user=> (take 10 (iterate inc 0))
(0 1 2 3 4 5 6 7 8 9)
user=> (take 10 (iterate (partial + 2) 0))
(0 2 4 6 8 10 12 14 16 18)
```

所谓序列是懒惰的，不仅意味着其仅在需要时（也可能永远不需要）才生成序列的尾元素，还意味着序列的头元素在使用后（如果不再需要使用）可以被舍弃。比如，下面的例子需要运行一段时间，但不会耗尽内存：

```
user=> (take-last 5 (range 0 100000000))
(99999995 99999996 99999997 99999998 99999999)
```

回到词频统计，get-pages返回的页面序列是懒惰的，因此count-words完全可以处理40 GiB的Wikipedia dump。另外这段代码很容易被并行化，明天将具体介绍。

第一天总结

第一天的学习结束了。第二天我们会对词频统计进行并行化，还要深入了解fold函数。

第一天我们学到了什么

由于普遍使用了共享可变状态，用命令式语言进行并发编程的难度是比较高的。函数式编程抛弃了共享可变状态，让并发编程变得更容易也更安全。本节中我们学习了以下知识：

- 用map或mapcat对一个序列的每个元素进行映射；
- 用序列的懒惰特性来处理较大的序列，甚至无穷序列；
- 用reduce将序列化简为一个（可能比较复杂的）值；
- 用fold对reduce进行并行化。

第一天自习

查找

- 阅读Clojure的cheat sheet，快速查阅Clojure的常用函数。
- 阅读lazy-seq的相关文档，使用lazy-seq可自建一个懒惰序列。

实践

- 与许多函数式语言不同，Clojure并不支持尾调用消除（tail-call elimination），因此Clojure代码通常很少使用递归。重写recursive-sum函数（参见本节“第一个函数式程序”部分），用loop和recur替换递归。
- 重写reduce-sum函数（参见本节“第一个函数式程序”部分），用读取器宏（reader macro）#()替换(fn ...)。

3.3　第二天：函数式并行

今天先来学习如何将Wikipedia词频统计程序并行化，然后再深入研究fold函数的细节，以说明如何用函数式编程来实现并行。

每次一页

第一天我们学习了map函数，其将某函数依次应用于输入序列的每个元素，并返回函数应用

后的新序列。但这个过程其实没有必要串行执行——Clojure提供了功能类似于map的pmap函数，其应用函数的过程是可以并行的。pmap在需要结果时可以并行计算，但仅生成所需要的而不是全部的结果，这个特性称为半懒惰（semi-lazy）[①]。

用下面的代码可以并行地将Wikipedia的页序列转换成词频map的序列：

```
(pmap #(frequencies (get-words %)) pages)
```

上述代码用读取器宏#(...)来声明传给pmap的函数，读取器宏可以快速创建匿名函数。函数的参数通过%1、%2等来标识，如果只有一个参数，可以通过%进行标识：

```
#(frequencies (get-words %))
```

与其等价的代码为：

```
(fn [page] (frequencies (get-words page)))
```

来测试一下：

```
wordcount.core=> (def pages ["one potato two potato three potato four"
           #_=>              "five potato six potato seven potato more"])
#'wordcount.core/pages
wordcount.core=> (pmap #(frequencies (get-words %)) pages)
({"one" 1, "potato" 3, "two" 1, "three" 1, "four" 1}
 {"five" 1, "potato" 3, "six" 1, "seven" 1, "more" 1})
```

现在可以将所得的序列化简成一个map，从而得到我们需要的词频总数。化简函数将按照以下规则合并两个map：

- 输出map的键是两个输入map的键的并集；
- 若某键存在于一个输入map中，那在输出map中的对应值等于在输入map中的对应值；
- 若某键存在于两个输入map中，那在输出map中的对应值等于在输入map中的对应值的和。

如图3-1所示。

我们可以自建一个化简函数，但（与以往一样）标准库已经提供了合适的函数。API文档如下：

```
(merge-with f & maps)
```

merge-with将maps中其余的map合并到第一个map中，返回合并后的map——如果多个map包含同一个键，那该键对应的多个值将被（从左向右地）合并到结果map中，合并的方法是调用(f val-in-result val-in-latter)。

① David Edgar Liebke给出过一个更容易理解的描述：http://incanter.org/downloads/fjclj.pdf。——译者注

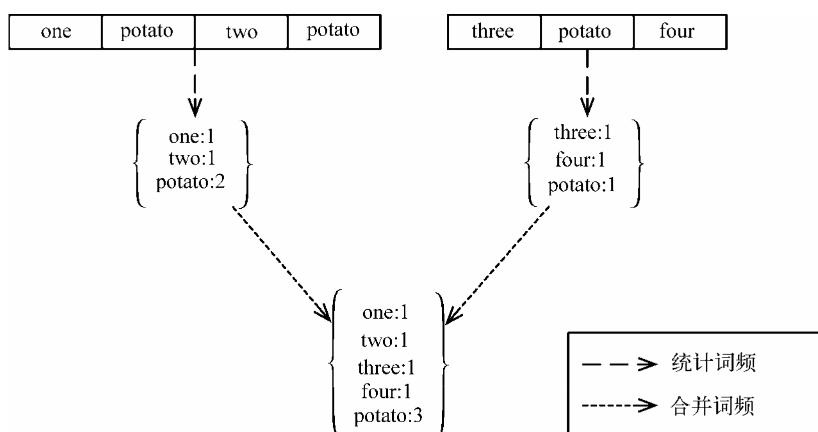


图3-1　将Wikipedia的两页合并成一个词频map

之前我们学习过`partial`函数,其返回一个被局部代入的函数,所以`(partial merge-with +)`可以返回一个用于合并map的函数，这个函数使用+对同一个键的多个值进行合并：

```
user=> (def merge-counts (partial merge-with +))
#'user/merge-counts
user=> (merge-counts {:x 1 :y 2} {:y 1 :z 1})
{:z 1, :y 3, :x 1}
```

将上述要点组装起来，就得到了并行版本的词频统计程序：

FunctionalProgramming/WordCount/src/wordcount/core.clj

```
(defn count-words-parallel [pages]
  (reduce (partial merge-with +)
    (pmap #(frequencies (get-words %)) pages)))
```

搞定，现在来测试一下性能。

利用批处理改善性能

在我的MacBook Pro上，用串行版本的词频统计程序处理Wikipedia的前100 000页需要花费140秒。并行版本需要花费94秒，加速比是1.5。我们已经使用并行得到了提速，但并不理想。

与之前在线程与锁方案中分析的原因类似（参见2.4节），逐页地进行计数和合并会导致大量的合并操作。如果能对页面进行批处理，将大大减少合并操作的次数，如图3-2所示。

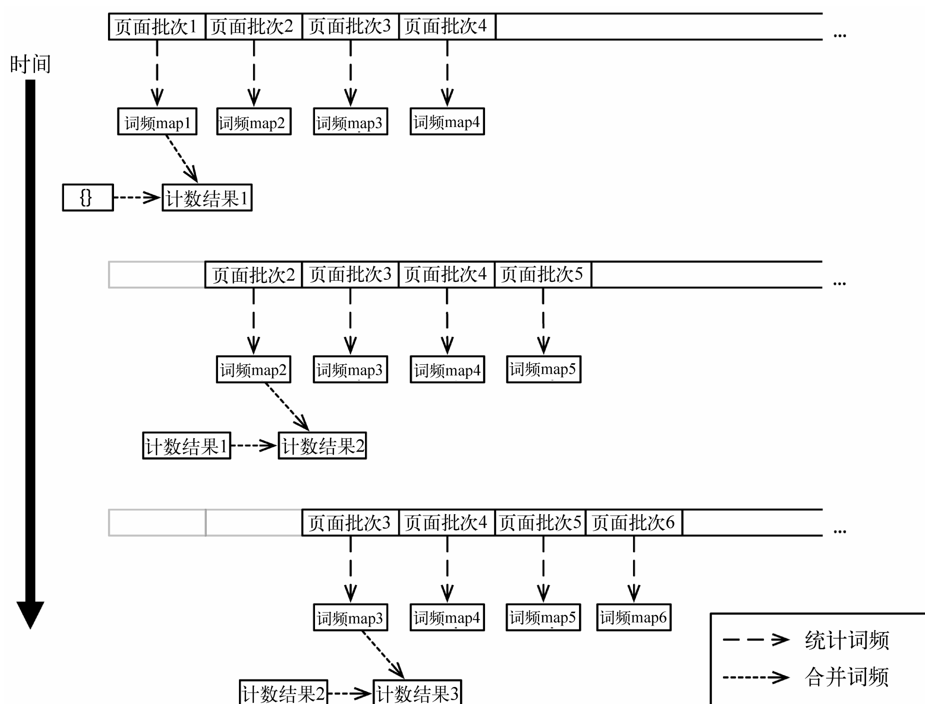


图3-2　批处理版本的词频统计

举例说明，将100个页面作为一个批次进行处理，`word-count`的代码修改如下：

FunctionalProgramming/WordCount/src/wordcount/core.clj

```
(defn count-words [pages]
  (reduce (partial merge-with +)
    (pmap count-words-sequential (partition-all 100 pages))))
```

这里用到了`partition-all`函数，其可以将一个序列中的元素分批（或称分区），构成多个序列：

```
user=> (partition-all 4 [1 2 3 4 5 6 7 8 9 10])
((1 2 3 4) (5 6 7 8) (9 10))
```

像之前一样，可以用`count-words-sequential`对每批页面进行词频统计，然后合并结果。不出所料，这一版本处理Wikipedia的前100 000页需要花费44秒，提速3.2倍。

化简器

第一天，我们曾经用`fold`替换`reduce`并获得了神奇的性能提升。想要理解`fold`的原理，需要先理解Clojure的`reducers`库。

化简器（reducer）描述了对集合进行化简的方法。普通版本的map接受一个函数和一个（可能是懒惰的）序列，并返回另一个（可能是懒惰的）序列：

```
user=> (map (partial * 2) [1 2 3 4])
(2 4 6 8)
```

而clojure.core.reducers提供的map则不同，接受相同的参数，但返回的是一个化简器reducible：

```
user=> (require '[clojure.core.reducers :as r])
nil
user=> (r/map (partial * 2) [1 2 3 4])
#<reducers$folder$reify__1599 clojure.core.reducers$folder$reify__1599@151964cd>
```

3

reducible不能作为值被直接使用，而是作为参数被传给reduce：

```
user=> (reduce conj [] (r/map (partial * 2) [1 2 3 4]))
[2 4 6 8]
```

上述代码中，conj函数的第一个参数是一个集合（初始时是空集合[]），其将第二个参数合并到第一个参数中。因此这段代码的结果与只执行map的结果相同。

into函数内部使用了reduce，所以下面这段代码与上面那段是等价的：


```
user=> (into [] (r/map (partial * 2) [1 2 3 4]))
[2 4 6 8]
```

clojure.core提供的大部分序列处理函数都有对应的化简器版本，包括之前见过的map和mapcat。与clojure.core提供的函数类似，其化简器版本也可以被嵌套使用：

```
user=> (into [] (r/map (partial + 1) (r/filter even? [1 2 3 4])))
[3 5]
```

一个化简器，并不代表函数返回的结果，而是代表如何产生结果的描述——被传给reduce或fold之前，化简器不会进行求值。这样做主要有两个好处：

- 嵌套的函数返回化简器比返回懒惰序列的效率更高，因为其不用构造处于中间状态的序列；
- 对整个嵌套链的集合操作，可以用fold进行并行化。

化简器内幕

为了理解化简器的原理，我们将创建一个略微简单却仍然高效的类似于clojure.core.reducers的库。首先需要理解Clojure的协议（protocol）。协议非常类似于Java中的接口——其是一系列方法的集合，并定义了一个抽象的概念。Clojure的集合就是通过CollReduce协议来支持化简操作的：

```
(defprotocol CollReduce
  (coll-reduce [coll f] [coll f init]))
```

CollReduce声明了一个函数coll-reduce，这是一个可变参数（multiple arities）函数——可以接受两个参数（coll、f），也可以接受三个参数（coll、f、init）。第一个参数类似于Java中的this，支持多态性分派（polymorphic dispatch）。来看看这段Clojure代码：

```
(coll-reduce coll f)
```

与这段代码等价的Java代码是：

```
coll.collReduce(f);
```

reduce函数只是简单地调用coll-reduce，而具体的任务执行则交给集合本身。我们自己实现的类似reduce函数中使用了这个特性：

FunctionalProgramming/Reducers/src/reducers/core.clj

```
(defn my-reduce
  ([f coll] (coll-reduce coll f))
  ([f init coll] (coll-reduce coll f init)))
```

这段代码还使用了一个之前没学过的defn特性——defn可以定义参数个数不同的多个函数（本例中是两个参数和三个参数）。my-reduce负责将参数转发给coll-reduce。来验证一下：

```
reducers.core=> (my-reduce + [1 2 3 4])
10
reducers.core=> (my-reduce + 10 [1 2 3 4])
20
```

现在来实现一个类似map的函数：

FunctionalProgramming/Reducers/src/reducers/core.clj

```
(defn make-reducer [reducible transformf]
  (reify
    CollReduce
    (coll-reduce [_ f1]
      (coll-reduce reducible (transformf f1) (f1)))
    (coll-reduce [_ f1 init]
      (coll-reduce reducible (transformf f1) init))))

(defn my-map [mapf reducible]
  (make-reducer reducible
    (fn [reducef]
      (fn [acc v]
        (reducef acc (mapf v))))))
```

这段代码定义了make-reducer函数，其接受一个化简器reducible和一个转换函数transformf，并返回一个CollReduce协议的实例。用reify实现一个协议，类似于在Java中用new创建一个接口的匿名实例。

这个CollReduce协议的实例会调用reducible的coll-reduce方法。其用transformf对f1进行转换，并用转换的结果（仍是一个函数）作为传给coll-reduce方法的一个参数。



小乔爱问：

下划线的作用？

在 Clojure 中下划线（_）通常作为未被使用的函数参数的参数名。之前的代码可以写成：

```
(coll-reduce [this f1]
  (coll-reduce reducible (transformf f1) (f1)))
```

但用下划线可以明确地表达出 this 是未被使用的。

对于传给make-reducer的转换函数transformf，其接受一个函数作为参数，并返回这个函数被转换后的版本。以my-map中的转换函数为例：

```
(fn [reducef]
  (fn [acc v]
    (reducef acc (mapf v))))
```

曾经介绍过，在化简过程中，为集合中的每个元素都会调用一次化简函数。化简函数的第一个参数是之前化简的结果（acc），第二个参数是集合中的某个元素。所以，在my-map中，对化简函数reducef的第二个参数需要用mapf（mapf是传给my-map的函数）进行转换。来验证一下：

```
reducers.core=> (into [] (my-map (partial * 2) [1 2 3 4]))
[2 4 6 8]
reducers.core=> (into [] (my-map (partial + 1) [1 2 3 4]))
[2 3 4 5]
```

当然，my-map也支持嵌套调用：

```
reducers.core=> (into [] (my-map (partial * 2) (my-map (partial + 1) [1 2 3 4])))
[4 6 8 10]
```

如果理解了上面的例子，就会发现其只进行了一次化简，化简函数是(partial * 2)和(partial + 1)组合后的函数。

我们已经学习了化简器是如何提供化简操作的。接下来将学习折叠操作fold是如何让化简操作并行化的。

分而治之

较之逐个元素地对集合进行化简，fold则使用二分算法。首先，fold（折叠操作）将集合分为两组，每组继续分为更小的两组，以此类推，直到每个分组的规模小于某个限制值（默认值是512）。其次，fold对每个分组进行逐个元素地化简。最后，对各分组的结果进行两两合并，直到剩下一个最终的结果。整个过程类似于一个二叉树，如图3-3所示。

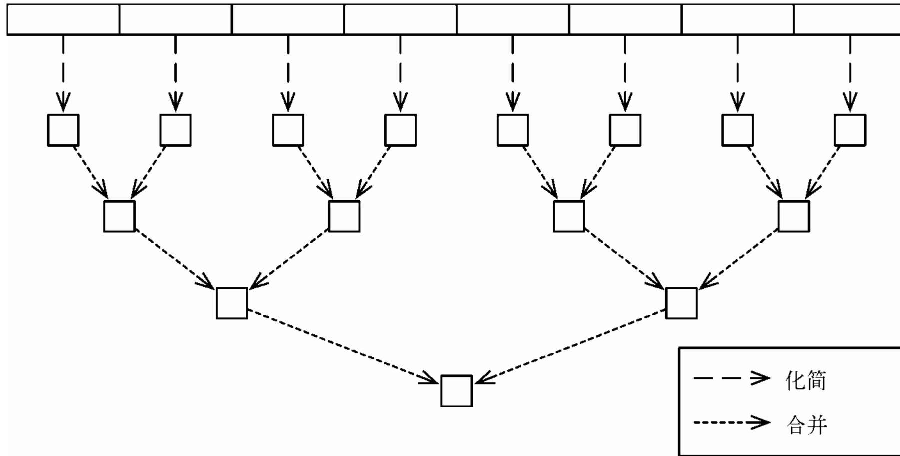


图3-3　fold二叉树

因为fold（利用Java 7的Fork/Join框架）创建了一个并行任务的处理树，化简操作和合并操作才得以并行执行。化简操作在树的叶子节点进行。下一级节点等待上一级节点的化简操作的结果，并将这些结果进行合并。整个过程将这样一直进行下去，直到树的根节点，获得最终的一组结果。

如果传给fold的函数仅有一个（之前的例子中是+），这个函数将被用于化简操作和合并操作。有时也需要让化简函数与合并函数不同，以后我们会学习到这一点。

对折叠的支持

可以进行折叠的集合不仅需要支持CollReduce协议，也需支持CollFold协议：

```
(defprotocol CollFold
  (coll-fold [coll n combinef reducef]))
```

类似于化简操作被委托给coll-reduce，折叠操作则被委托给coll-fold：

FunctionalProgramming/Reducers/src/reducers/core.clj

```
(defn my-fold
  ([reducef coll]
    (my-fold reducef reducef coll))
  ([combinef reducef coll]
    (my-fold 512 combinef reducef coll))
  ([n combinef reducef coll]
    (coll-fold coll n combinef reducef)))
```

当接受两个参数或三个参数时，my-fold为combinef和n提供了默认值，并调用自己。当接受四个参数时，my-fold将调用集合的coll-fold函数。

为了让make-reducer支持折叠操作，只需要让make-reducer实现CollReduce的同时再实现CollFold：

FunctionalProgramming/Reducers/src/reducers/core.clj

```
(defn make-reducer [reducible transformf]
  (reify
➤   CollFold
➤   (coll-fold [_ n combinef reducef]
➤     (coll-fold reducible n combinef (transformf reducef)))

    CollReduce
    (coll-reduce [_ f1]
      (coll-reduce reducible (transformf f1) (f1)))
    (coll-reduce [_ f1 init]
      (coll-reduce reducible (transformf f1) init))))
```


实现CollFold非常类似于实现CollReduce——首先对参数中的化简函数进行转换，然后将参数传给reducible的coll-fold。来验证一下：

```
reducers.core=> (def v (into [] (range 10000)))
#'reducers.core/v
reducers.core=> (my-fold + v)
49995000
reducers.core=> (my-fold + (my-map (partial * 2) v))
99990000
```

下面这个例子使用了不同函数分别进行化简和合并。

用折叠实现词频统计

回到之前词频统计的场景，用fold来实现frequencies函数时，非常适合使用不同的函数进行化简和合并：

FunctionalProgramming/Reducers/src/reducers/parallel_frequencies.clj

```
(defn parallel-frequencies [coll]
  (r/fold
    (partial merge-with +)
    (fn [counts x] (assoc counts x (inc (get counts x 0))))
    coll))
```

这让我们想起了今天早些时候学习的批处理版本的count-words——每一批次都化简成map，然后通过(partial merge-with +)进行合并。

不过由于fold不能适用于懒惰序列（因为无法对懒惰序列进行二分），因此没办法用Wikipedia的页来测试fold。但可以用一个很长的随机数序列来进行模拟测试。

repeatedly函数反复调用传入的函数来构造一个无穷的懒惰序列。本例中传入的是rand-int函数，每次调用rand-int会得到一个随机整数：

```
user=> (take 10 (repeatedly #(rand-int 10)))
(2 6 2 8 8 5 9 2 5 5)
```

用下面的代码可以构造一个很长的随机数序列：

```
reducers.core=> (def numbers (into [] (take 10000000 (repeatedly #(rand-int 10)))))
#'reducers.core/numbers
```

现在分别用frequencies和parallel-frequencies对该随机数序列的整数频率进行统计：

```
reducers.core=> (require ['reducers.parallel-frequencies :refer :all])
nil
reducers.core=> (time (frequencies numbers))
"Elapsed time: 1500.306 msecs"
{0 1000983, 1 999528, 2 1000515, 3 1000283, 4 997717, 5 1000101, 6 999993, …
reducers.core=> (time (parallel-frequencies numbers))
"Elapsed time: 436.691 msecs"
{0 1000983, 1 999528, 2 1000515, 3 1000283, 4 997717, 5 1000101, 6 999993, …
```

可以看到串行执行的frequencies运行了1500 ms，而并行版本运行了400+ ms，提速近3.5倍。

第二天总结

我们在第二天中讨论了如何用Clojure实现并行。明天将关注如何用future模型和promise模型实现并行，以及如何利用它们进行数据流式编程（dataflow programming）。

第二天我们学到了什么

Clojure可以将串行操作轻松自然地并行化。

- pmap可以将映射操作并行化，构造一个半懒惰的map。
- 利用partition-all可以对并行的映射操作进行批处理，以提高处理效率。
- fold使用分而治之的策略，可以将化简操作并行化。
- clojure.core.reducers包提供的类似map、类似mapcat、类似filter的函数返回的并不是序列，而是化简器reducible，可以说这是化简操作的关键所在。

第二天自习

查找

- 观看Rich Hickey在QCon 2012上介绍reducers库的视频。
- 阅读pcalls和pvalues的相关文档——这两个函数与pmap有什么区别？是否能利用pmap来实现它们？

实践

- 在my-map的基础上创建my-flatten和my-mapcat函数。注意：它们都比my-map更复杂，因为需要将输入序列的一个元素对应到输出序列的一个或多个元素。遇到困难时可以参见本书配套代码。
- 创建my-filter函数。它也比my-map更复杂，因为需要减少输入序列的元素个数。

3.4 第三天：函数式并发

前两天我们一直在关注并行，今天会将注意力转向并发。在这之前，我们将进一步探究为何函数式编程能轻易地实现并行化。

同样的结构，不同的求值顺序

回顾过去两天，我们的学习一直围绕着同一个主题——使用函数式编程可以玩转程序的求值顺序。如果两个计算过程相互独立，就可以任意安排这两个计算过程的求值顺序，包括让它们并行。

下面的代码，均进行相同的计算、返回同样的结果、具有相似的代码结构，但它们以完全不同的顺序进行求值：

```
(reduce + (map (partial * 2) (range 10000)))
```

这段代码化简一个嵌套了懒惰序列的懒惰序列——每个懒惰序列的元素都是按需求值的。

```
(reduce + (doall (map (partial * 2) (range 10000))))
```

这段代码首先构造map的输出序列（doall强迫懒惰序列对全部元素进行求值），然后再进行化简。

```
(reduce + (pmap (partial * 2) (range 10000)))
```

这段代码化简一个半懒惰的序列，这个序列是并行求值的。

```
(reduce + (r/map (partial * 2) (range 10000)))
```

这段代码用单个化简函数对一个懒惰序列进行化简，该化简函数由+和(partial * 2)组合而成。

```
(r/fold + (r/map (partial * 2) (into [] (range 10000))))
```

这段代码首先构造range的输出序列，然后创建进行化简和合并的处理树，并根据此树对序列进行并行地化简。

在Java这类命令式语言中，求值顺序与源码的语句顺序紧密相关。虽然编译器和运行时（runtime）都可能造成一些乱序执行（比如我们在讨论线程与锁时一直在留意的乱序执行现象，参见2.2节中“诡异的内存”部分），但一般来说，求值顺序与其在代码中的顺序基本一致。

函数式语言则更有一种*声明式*的范儿。函数式程序并不是描述“如何求值以得到结果”，而是描述“结果应当是什么样的”。因此，在函数式编程中，如何安排求值顺序来获得最终结果是相对自由的，这正是函数式代码可以轻松并行的关键所在。

下一节我们将学习为什么函数式编程可以玩转求值顺序，而命令式语言却不具有这种能力。

引用透明性

在纯粹的函数式语言中，函数都具有引用透明性——在任何调用函数的地方，都可以用函数运行的结果来替换函数的调用，而不会对程序产生副作用。来看一些例子：

```
(+ 1 (+ 2 3))
```

它与下面的代码是等价的：

```
(+ 1 5)
```

其实，描述函数式代码执行方式的一种方法就是不断用函数的执行结果替换函数的调用，直到得到最终结果。例如按照下面的方式来计算`(+ (+ 1 2) (+ 3 4))`：

```
(+ (+ 1 2) (+ 3 4)) → (+ (+ 1 2) 7) → (+ 3 7) → 10
```

也可以是下面的求值顺序：

```
(+ (+ 1 2) (+ 3 4)) → (+ 3 (+ 3 4)) → (+ 3 7) → 10
```

当然，这个结论对于Java的+操作也适用。与Java不同的是，函数式编程的**每个**函数都具有引用透明性。这也是我们能安全地调整函数求值顺序的关键所在。



小乔爱问：

Clojure不是不纯粹吗？

下一章我们将学习到 Clojure 是一门不纯粹的函数式语言——在 Clojure 中可以构造带有副作用的函数，这样的函数不具有引用透明性。

在实践中这并不会造成很大影响，因为常见的 Clojure 代码极少出现副作用，并且出现副作用时会非常明显。有一些规则描述了副作用在何处出现是安全的，只要遵循这些规则就不大可能遭遇由求值顺序带来的问题。

数据流

我们来思考一下数据是如何在函数间流动的。图3-4示意的是`(+ (+ 1 2) (+ 3 4))`的数据流。

由于`(+ 1 2)`和`(+ 3 4)`之间没有依赖关系，所以理论上这两步求值能以任何顺序进行，包括同时执行。前两步求值得到结果后，最后一步加法才能进行。

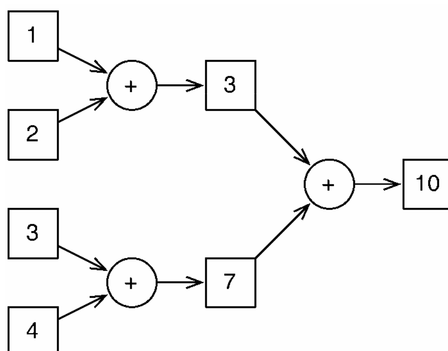


图3-4 (+ (+ 1 2) (+ 3 4))的数据流

理论上，运行时可以从这幅图的左端出发，向右端“推进”数据。当某函数所依赖的数据都可用时，该函数就可以执行了。所有函数（至少是理论上）都可以同时执行。这种执行方式被称为数据流式编程（dataflow programming）。Clojure提供了future模型和promise模型来支持这种执行方式。

Future模型

future函数可以接受一段代码，并在一个单独的线程中执行这段代码。其返回一个future对象：

```
user=> (def sum (future (+ 1 2 3 4 5)))
#'user/sum
user=> sum
#<core$future_call$reify__6110@5d4ee7d0: 15>
```

可以利用deref或其简写@对future对象进行解引用，来获取其代表的值：

```
user=> (deref sum)
15
user=> @sum
15
```

对future对象进行解引用将阻塞当前线程，直到其代表的值变得可用（或者称为被求值）。现在我们可以准确地构造出图3-4所示的数据流图了：

```
user=> (let [a (future (+ 1 2))
  #_=>       b (future (+ 3 4))]
  #_=>   (+ @a @b))
10
```

这段代码首先用let将(future (+ 1 2))赋给a，并将(future (+ 3 4))赋给b。对(+ 1 2)和(+ 3 4)的求值可以分别在不同线程中进行。最后，外层的加法将一直阻塞，直到内层的加法完成。

当然，对于这种两个数求和的简单计算，使用future模型是大材小用——之后我们还将学习更有现实意义的例子。不过还是先来了解一下Clojure的promise模型。

Promise模型

类似于future对象，promise对象也是异步求值的，也通过deref或@解引用，在求值前也会阻塞线程。不同的是创建一个promise对象后，使用promise对象的代码并不会立刻执行，而是等到用deliver为promise对象赋值后才会执行。下面用一个REPL会话来举例：

```
user=> (def meaning-of-life (promise))
#'user/meaning-of-life
user=> (future (println "The meaning of life is:" @meaning-of-life))
#<core$future_call$reify__6110@224e59d9: :pending>
user=> (deliver meaning-of-life 42)
#<core$promise$reify__6153@52c9f3c7: 42>
The meaning of life is: 42
```

首先，构造了一个叫meaning-of-life的promise对象。然后，用future函数创建一个线程来打印其值（像这样利用future函数创建线程是Clojure的惯例）。最后，用deliver为promise对象赋值，之前创建的线程就不再阻塞了。

我们已经学习了future模型和promise模型，现在来用它们创建一个真实的应用。

函数式Web服务

我们将创建一个Web服务，用来接收实时的文本数据（例如，一个电视节目的脚本），并进行翻译。文本数据由片段（snippet）构成，片段都带有序号。以路易斯·卡罗尔的诗作*Jabberwocky*（选自《爱丽丝镜中奇遇》）的第一节为例，来说明片段这个概念：

0 Twas brillig, and the slithy roves
1 Did gyre and gimble in the wabe:
2 All mimsy were the borogoves,
3 And the mome raths outgrabe.

如果要将片段0提交到Web服务，就要构造一个发往*/snippet/0*的PUT请求，其内容是“Twas brillig, and the slithy roves”。片段1将被发往*/snippet/1*，以此类推。

这是一个非常简单的API，然而实现起来并不像看上去那么简单。首先，代码是运行在一个并发的Web服务器上的，这就要求代码是线程安全的。其次，由于网络的特性，代码需要处理一些特殊情况，例如片段丢失、重试、重复提交和乱序提交。

如果需要按序号处理片段（即片段的处理与片段到达服务器的时间先后无关），就必须要记录哪些片段已经被接收、哪些片段已经被处理。当接收到新的片段时，需要检查是否可以继续处

理片段。这个任务并不容易，值得我们秀一下如何用并发来构造一个简单的解决方案。

图3-5展示了解决方案。

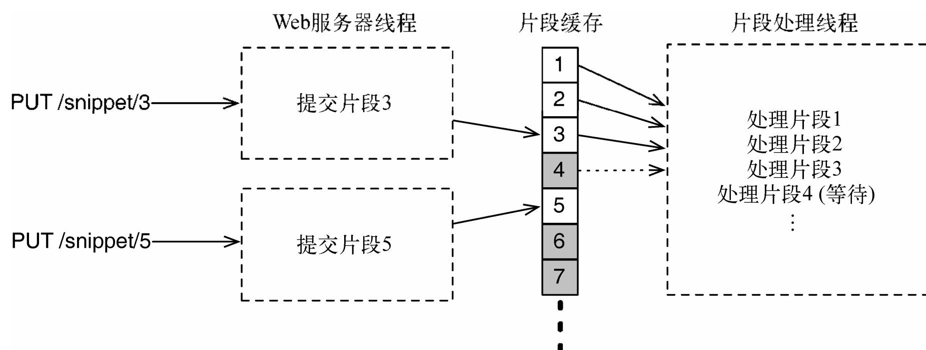


图3-5　文本数据处理流程

图3-5的左端是Web服务器创建的线程，用来处理输入请求；右端是串行处理片段的线程，正在等待下一个可用片段。下一节将讨论`snippets`结构，其将被用于线程之间的交互。

接收片段

我们用下面的结构来记录已经被接收的片段：

FunctionalProgramming/TranscriptHandler/src/server/core.clj

```
(def snippets (repeatedly promise))
```

`snippets`是一个由promise对象构成的无穷懒惰序列。当某个片段可用时，调用`accept-snippet`来为对应的promise对象赋值：

FunctionalProgramming/TranscriptHandler/src/server/core.clj

```
(defn accept-snippet [n text]
  (deliver (nth snippets n) text))
```

要串行地处理片段，只需创建一个线程，按序号对每个promise对象进行解引用即可。例如，下面的代码可以串行输出每个片段的值：

FunctionalProgramming/TranscriptHandler/src/server/core.clj

```
(future
  (doseq [snippet (map deref snippets)]
    (println snippet)))
```

`doseq`会串行处理一个序列。本例中，`doseq`处理的序列是由解引用后的promise对象构成的，`snippet`指向正在处理的元素。

剩下的工作就是将所有这些组装成一个Web服务。下面的代码利用了Compojure库[1]：

FunctionalProgramming/TranscriptHandler/src/server/core.clj

```
(defroutes app-routes
  (PUT "/snippet/:n" [n :as {:keys [body]}]
    (accept-snippet (edn/read-string n) (slurp body))
    (response "OK")))

(defn -main [& args]
  (run-jetty (site app-routes) {:port 3000}))
```

这段代码定义了一个PUT路由，其将调用`accept-snippet`函数。我们使用了内置的Web服务器Jetty[2]——与大多数Web服务器类似，Jetty是多线程的，因此要求代码是线程安全的。

现在可以用`lein run`启动该服务器，并通过`curl`命令做一些验证。比如发送片段0：

```
$ curl -X put -d "Twas brillig, and the slithy toves" \
-H "Content-Type: text/plain" localhost:3000/snippet/0
OK
```

服务器马上输出：

```
Twas brillig, and the slithy toves
```

如果在发送片段1之前发送片段2，那么就不会有任何输出：

```
$ curl -X put -d "All mimsy were the borogoves," \
-H "Content-Type: text/plain" localhost:3000/snippet/2
OK
```

接着发送片段1：

```
$ curl -X put -d "Did gyre and gimble in the wabe:" \
-H "Content-Type: text/plain" localhost:3000/snippet/1
OK
```

现在片段1和片段2都会被输出：

```
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
```

多次发送同一个片段是没有问题的，因为当promise对象已经被赋值后，再次调用`deliver`将不会触发任何动作。所以下面的命令不会带来任何问题，也不会有任何输出：

```
$ curl -X put -d "Did gyre and gimble in the wabe:" \
-H "Content-Type: text/plain" localhost:3000/snippet/1
OK
```

至此，我们已经学习了如何处理片段，下面将做一些更有趣的尝试。设想有另外一个用于翻

① https://github.com/weavejester/compojure

② http://www.eclipse.org/jetty/

译文本的Web服务，来修改一下代码，使用新的Web服务进行文本翻译。

句子

在学习如何调用翻译服务之前，要先将片段的序列转换成句子的序列。句子的分隔符可能出现在片段的任意位置，所以需要对片段进行分割或合并，以解析出句子。

首先，按照句子的分隔符进行分割：

FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj

```
(defn sentence-split [text]
  (map trim (re-seq #"[^\.!\?:;]+[\.!\?:;]*" text)))
```

re-seq接受一个正则表达式来匹配句子，并返回匹配的序列。可以用trim删除不必要的空格：

```
server.core=> (sentence-split "This is a sentence. Is this?! A fragment")
("This is a sentence." "Is this?!" "A fragment")
```

接下来，使用一个正则表达式的技巧以判断某字符串是不是一个完整的句子：

FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj

```
(defn is-sentence? [text]
  (re-matches #"^.*[\.!\?:;]$" text))

server.core=> (is-sentence? "This is a sentence.")
"This is a sentence."
server.core=> (is-sentence? "A sentence doesn't end with a comma,")
nil
```

最后，将这些知识点组装在一起，创建strings->sentences函数。该函数接受一个字符串序列，并返回一个句子序列：

FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj

```
(defn sentence-join [x y]
  (if (is-sentence? x) y (str x " " y)))

(defn strings->sentences [strings]
  (filter is-sentence?
    (reductions sentence-join
      (mapcat sentence-split strings))))
```

这里用到了reductions函数。顾名思义，reductions函数与reduce类似，唯一的区别是它不返回单一值，而是返回由每一步骤的中间值构成的序列：

```
server.core=> (reduce + [1 2 3 4])
10
server.core=> (reductions + [1 2 3 4])
(1 3 6 10)
```

在此使用了sentence-join作为化简函数。如果第一个参数是个完整的句子，就返回第二个

参数；否则，就返回将两个参数（用空格）连接后的结果：

```
server.core=> (sentence-join "A complete sentence." "Start of another")
"Start of another"
server.core=> (sentence-join "This is" "a sentence.")
"This is a sentence."
```

与`reductions`合用时：

```
server.core=> (def fragments ["A" "sentence." "And another." "Last" "sentence."])
#'server.core/fragments
server.core=> (reductions sentence-join fragments)
("A" "A sentence." "And another." "Last" "Last sentence.")
```

用`is-sentense?`过滤结果：

```
server.core=> (filter is-sentence? (reductions sentence-join fragments))
("A sentence." "And another." "Last sentence.")
```

这样就得到了句子序列，现在可以将其传给负责翻译的服务器了。

翻译句子

使用future模型的一个典型场景是与其他服务器之间的通信。future模型允许主线程运行时，将访问网络之类的操作放在另一个线程上进行。下面是`translate`函数，其返回一个future对象，对这个future对象求值将获得函数参数翻译后的结果：

FunctionalProgramming/TranscriptHandler2/src/server/core.clj

```
(def translator "http://localhost:3001/translate")

(defn translate [text]
  (future
    (:body (client/post translator {:body text}))))
```

这段代码用到了clj-http库[①]提供的函数`client/post`，来进行POST请求并获取返回。现在可以使用`translate`函数，对之前`strings->sentences`的结果进行翻译，其结果是一个集合。

FunctionalProgramming/TranscriptHandler2/src/server/core.clj

```
(def translations
  (delay
    (map translate (strings->sentences (map deref snippets)))))
```

这里用到了`delay`函数，其创建一个懒惰的值——在被解引用前不会进行求值。

组装

下面是文本翻译Web服务的完整代码：

① https://github.com/dakrone/clj-http

FunctionalProgramming/TranscriptHandler2/src/server/core.clj

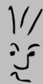
```
Line 1 (def snippets (repeatedly promise))
     -
     - (def translator "http://localhost:3001/translate")
     -
     5 (defn translate [text]
     -   (future
     -     (:body (client/post translator {:body text}))))
     -
     - (def translations
    10   (delay
     -     (map translate (strings->sentences (map deref snippets)))))
     -
     - (defn accept-snippet [n text]
     -   (deliver (nth snippets n) text))
    15
     - (defn get-translation [n]
     -   @(nth @translations n))
     -
     - (defroutes app-routes
    20   (PUT "/snippet/:n" [n :as {:keys [body]}]
     -     (accept-snippet (edn/read-string n) (slurp body))
     -     (response "OK"))
     -   (GET "/translation/:n" [n]
     -     (response (get-translation (edn/read-string n)))))
    25
     - (defn -main [& args]
     -   (run-jetty (wrap-charset (api app-routes)) {:port 3000}))
```

这段代码中添加了一个GET入口，用来获取翻译的结果（第23行）。其中使用了get-translation函数（第16行），用来访问translations序列。

现在可以运行一下我们的成果了。首先启动上面创建的服务器，然后启动本书配套代码中的翻译服务器，最后运行TranscriptTest程序（同样也在配套代码中），现在你应当能看到诗歌*Jabberwocky*被逐句翻译成法语：

```
$ lein run
Il brilgue, les tôves lubricilleux Se gyrent en vrillant dans le guave:
Enmîmés sont les gougebosqueux Et le mômerade horsgrave.
Garde-toi du Jaseroque, mon fils!
La gueule qui mord; la griffe qui prend!
Garde-toi de l'oiseau Jube, évite Le frumieux Band-à-prend!
«...»
```

至此我们达成了目标——一个综合运用了懒惰性、future模型、promise模型的完整的并发Web服务器。其中没有使用可变状态，也没有使用锁。比起用命令式语言实现的等价解决方案，我们的方案更简单易读。



小乔爱问：

我们是否一直持有序列的头元素？

上面的 Web 服务使用了两个懒惰序列：`snippets` 和 `translations`。程序一直在持有这两个序列的头元素（参见 3.2 节中“懒惰一点好”部分），因此这两个序列将一直增长。程序也将占用越来越多的内存。

下一章我们将学习如何用 Clojure 的引用类型来解决这个隐患，并修改 Web 服务，让其可以处理多个文本的数据。

第三天总结

我们结束了第三天的学习。这些天讨论了如何用函数式编程高效地实现并行和并发。

第三天我们学到了什么

函数式编程中的函数具有引用透明性。利用这个特性可以安全地对函数的求值顺序进行调整，而不会影响到程序的运行。值得一提的是，利用这个特性可以让代码在其所依赖的数据被准备好时才可运行，这也称为数据流式编程（Clojure提供future模型和promise模型对其进行支持）。我们还通过一个例子，用数据流式编程简化了Web服务的实现。

第三天自习

查找

- `future`与`future-call`有什么区别？如何用其中一个实现另一个？
- 如何鉴别一个future对象被求值时是否进行了阻塞？如何取消一个future对象？

实践

- 修改之前创建的文本处理服务器，处理发往/translation/:n的GET请求时，如果暂时还没有翻译结果，就不进行阻塞，而是返回状态码409。
- 用命令式语言实现文本处理服务器。其是否与函数式版本一样简洁？如何保证它不存在竞态条件？

3.5　复习

许多人对并行的理解存在一个误区——认为并行一定会伴随着不确定性，如果不串行执行，那么我们就不能依赖某一种执行顺序的结果，必须时刻警惕竞态条件。

当然，有一些并发程序一定会带有不确定性。这对它们来说是不可避免的——有一些场景天生就依赖于时序。但这并不意味着所有的并行程序都有不确定性。例如，对0到10 000之间的数求和，即使将串行加法改为并行加法，也不会改变结果；无论用多少线程对某个Wikipedia dump进行词频统计，其结果总是相同的。

在使用线程与锁模型的程序中，大多数潜藏的竞态条件并不是来自于问题本身的不确定性，而是隐藏于解决方案的细节中。

函数式代码具有引用透明性，因此可以随意改变其执行顺序，而不会对最终结果产生影响。我们可以顺理成章地让相互独立的函数并行执行——本章的例子也利用这个特性轻易地将函数式代码并行化了。



小乔爱问：

为什么没看到单子（Monad）和幺半群（Monoid）？

通常介绍函数式编程时都会对一些数学概念进行阐述，例如单子、幺半群、范畴论（Category theory）。我们用了一整章来介绍函数式编程，却没有涉及这些概念。为什么？

程序员对编程语言的偏好很大程度上取决于语言的类型系统。使用Java、Scala之类的静态类型语言的体验，与使用Ruby、Python之类的动态类型语言的体验是完全不同的。

静态类型语言强迫程序员在早期必须选择正确的类型。只有付出这样的代价，编译器才能确保运行时不发生类型错误，并且类型系统可以优化执行效率。

动态类型语言不强迫程序员在早期付出如此代价，但程序员要承担运行时发生类型错误或者运行效率较低的风险。

在函数式编程的范畴也同样存在这种分歧。像Haskell这种静态类型的函数式语言利用单子和幺半群等数学概念为类型系统增加了以下能力：明确限制了某些函数和某些值可以使用的位置，在保持函数性的同时可以检测代码的副作用。

在使用Clojure时，学习这些数学概念对理解理论无疑非常有帮助，但Clojure使用的不是静态类型系统，因此介绍这些数学概念的实用意义不大。另一方面，由于编译器不会对相关的错误进行告警，因此程序员必须手工确认函数和值的使用场景是正确的，这无疑增加了程序员的负担。

优点

使用函数式编程最大的好处是我们可以确信程序是按照我们预想的方式运行的。一旦上手（这可能需要花些时间，如果你对命令式编程“中毒”已深则更是如此），比起等价的命令式程序，

函数式程序会更简单，更容易推理，也更便于测试。

如果我们写了一个函数式解决方案，利用函数式程序的引用透明性，我们可以轻松地将程序并行化，或者在一个并发环境下使用该方案。由于函数式代码不使用可变状态，大部分存在于线程与锁模型中的并发bug将销声匿迹。

缺点

很多人认为函数式代码比起等价的命令式代码效率较低。对于某些场景确实存在性能损失，但大部分场景性能损失是远低于预期的。而且用少许性能损失来换取程序健壮性和扩展性的提升是值得的。

其他语言

近期，Java 8添加了一系列新特性，使用这些特性可以更容易地写出函数式代码，其中最有名的是lambda表达式[①]和stream API[②]。stream API支持聚合操作，其类似于Clojure的reducer，可以并行地处理流。

所有的函数式编程都会介绍Haskell[③]。本章和之后的章节介绍的所有技术在Haskell都可以找到。Simon Marlow的教程[④]是学习Haskell并行编程和并发编程的绝佳选择。

结语

本章介绍了函数式编程在并发与并行方面的诸多优势，然而其优点还远不止于此。可以断言函数式编程在未来会变得更重要。

前面已经提到过，在可预见的将来，可变状态会一直伴随在我们左右。下一章我们将学习Clojure如何处理函数的副作用，同时又不牺牲对并发的支持。

---

① http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
② http://docs.oracle.com/javase/tutorial/collections/streams/index.html
③ http://haskell.org/
④ http://community.haskell.org/~simonmar/par-tutorial.pdf

第 4 章

Clojure之道——分离标识与状态

4

现代的混合动力客车同时拥有内燃机和电动机的优点。根据环境不同，有时只使用汽油，有时只使用电力，有时则同时使用两者。与之类似，Clojure也提供了一种方法，混合了函数式编程和可变状态——这种方法平衡了两者优点，成为并发编程的利器。

4.1 混搭的力量

函数式编程对于某些问题是非常适用的，但对于某些状态易变的问题则不然。虽然函数式编程可以用来解决这类问题，但用传统的思路处理会更加容易一些。上一章中介绍了Clojure的纯函数子集，而本章中我们将偏离之前的内容，学习如何用Clojure提供的混搭技术来解决这类问题。

第一天，我们将讨论原子变量，它是Clojure提供的用于并发的可变数据类型。我们也将学习如何使用原子变量和持久数据结构来分离标识与状态。第二天，学习Clojure的其他可变数据结构：代理和软件事务内存。第三天，将分别用原子变量和STM实现一个算法，并讨论两种解决方案的利弊。

4.2 第一天：原子变量与持久数据结构

纯粹的函数式语言完全不支持可变量[①]。相比之下，Clojure是不纯粹的——其为不同的并发场景提供了大量的多样的可变量。通过使用可变量和持久数据结构（我们稍后会解释持久的意思），可以避开传统并发程序中共享可变状态带来的诸多问题。

---

① 在此将mutable variables/data译为“可变量”，意为这些“量”在整个生命周期中其值是可能改变的。在数学领域中，应称其为“变量”。但“变量”一词在编程领域中指的是“编译期结束后其值可能改变的量”，与数学领域中“整个生命周期中其值是可能改变的量”有少许差异，为避免混淆，将其译为“可变量”。同样的差异也存在于“常量”（编译期就能确定其值不改变的量）和“不变量”（整个生命周期中其值不改变的量）。另外，在此将variable和data统称为“量”，是为了简化相关的概念，不会影响之后的阅读。——译者注

在此要特别强调不纯粹的函数式语言与命令式语言的区别。在命令式语言中，变量默认都是状态易变的，代码会经常修改变量。而在不纯粹的函数式语言中，变量默认是状态不易变的，代码仅在十分必要时才修改变量。稍后我们会学习到：使用Clojure的可变量，可以在保证安全性和数据一致性的同时，处理好可变状态带来的副作用。

今天我们要学习如何使用可变量和持久数据结构来分离标识与状态。采用这些技术，多线程可以不使用锁（当然也就不会有死锁的风险）访问可变量，同时也不会碰到3.2节中介绍的风险（隐藏可变状态和逃逸可变状态）。先来看看Clojure提供的最简单的可变量类型——*原子变量*。

原子变量

原子变量就是具有原子性的变量，非常类似于2.3节中介绍的原子变量（事实上Clojure的原子变量就是在`java.util.concurrent.atomic`的基础上建立的）。下面的例子用于创建原子变量并获取其值：

```
user=> (def my-atom (atom 42))
#'user/my-atom
user=> (deref my-atom)
42
user=> @my-atom
42
```

使用`atom`函数可以创建原子变量，其参数是原子变量的初始值。通过`defer`或`@`可以获得原子变量的值。

使用`swap!`可以更新原子变量的值：

```
user=> (swap! my-atom inc)
43
user=> @my-atom
43
```

`swap!`接受一个函数，并将原子变量的当前值传给该函数，该函数的返回值将作为原子变量的新值。也可以将额外的参数传给函数，例如：

```
user=> (swap! my-atom + 2)
45
```

传给函数的第一个参数是原子变量的当前值，如果`swap!`有额外参数，则会依次传给该函数。本例中，原子变量的新值是`(+ 43 2)`。

一个不太常用的函数是`reset!`，可以用来重置原子变量的值，无论原子变量是什么值：

```
user=> (reset! my-atom 0)
0
user=> @my-atom
0
```

原子变量可以是任何类型——很多Web应用都是用原子map来存储会话数据的，例如：

```
user=> (def session (atom {}))
#'user/session
user=> (swap! session assoc :username "paul")
{:username "paul"}
user=> (swap! session assoc :session-id 1234)
{:session-id 1234, :username "paul"}
```

我们已经通过REPL了解了原子变量的一些特性，现在来看一个实际应用的例子。

具有可变状态的多线程Web服务

3.2节讨论了一个假想的用于管理比赛的Web服务。这一节我们会完整实现这个Web服务，并学习如何使用Clojure的持久数据结构来避免Java中逃逸可变状态的风险。

Clojure/TournamentServer/src/server/core.clj

```
Line 1 (def players (atom ()))
     -
     - (defn list-players []
     -   (response (json/encode @players)))
     5
     - (defn create-player [player-name]
     -   (swap! players conj player-name)
     -   (status (response "") 201))
     -
    10 (defroutes app-routes
     -   (GET "/players" [] (list-players))
     -   (PUT "/players/:player-name" [player-name] (create-player player-name)))
     - (defn -main [& args]
     -   (run-jetty (site app-routes) {:port 3000}))
```

这段代码定义了两个路由——发往`/players`的GET请求会返回当前的运动员列表（JSON格式），发往`/players/name`的PUT请求则会添加一个运动员。与上一章的Web服务一样，由于使用的Jetty服务器是多线程的，因此需要保证代码是线程安全的。

在讨论这段代码的工作原理之前，先来直接感受一下。可以在命令行中调用`curl`进行测试：

```
$ curl localhost:3000/players
[]

$ curl -X put localhost:3000/players/john
$ curl localhost:3000/players
["john"]

$ curl -X put localhost:3000/players/paul
$ curl -X put localhost:3000/players/george
$ curl -X put localhost:3000/players/ringo
$ curl localhost:3000/players
["ringo","george","paul","john"]
```

现在来看看core.clj的工作原理。原子变量`players`（第1行）被初始化成空列表`()`。通过`conj`可以添加新的运动员（第7行），并返回HTTP状态201（表示已创建）的空响应。通过@获取`players`的值，并以JSON形式返回运动员列表（第4行）。

一切看上去很简单（实际上也确实如此），但有一件事情困扰着我们。`list-players`和`create-player`函数都访问`players`——为什么这段代码不会有之前逃逸可变状态的问题？如果一个线程正在遍历`players`并将其转换为JSON格式，而另一个线程同时向`players`中添加元素，那么会发生什么？

Clojure的数据结构是持久的，因此这段代码才是线程安全的。

持久数据结构

我们这里说的“持久”并不是指将数据持久化到磁盘或者保存到数据库中，而是指数据结构被修改时总是保留其之前的版本，这样可以为代码提供一致的数据视角。来用REPL看一个简单的例子：

```
user=> (def mapv1 {:name "paul" :age 45})
#'user/mapv1
user=> (def mapv2 (assoc mapv1 :sex :male))
#'user/mapv2
user=> mapv1
{:age 45, :name "paul"}
user=> mapv2
{:age 45, :name "paul", :sex :male}
```

持久数据结构被修改时看上去就像创建了一个完整的副本。如果持久数据结构在实现时也创建完整副本，那将非常低效并且使用限制很大（可以类比2.4节介绍过的`CopyOnWriteArrayList`）。幸运的是，持久数据结构的实现选择了更精巧的方法，其中使用了*共享结构*。

最容易理解的持久数据结构就是列表。来看一个简单的例子：

```
user=> (def listv1 (list 1 2 3))
#'user/listv1
user=> listv1
(1 2 3)
```

图4-1展示了上述列表在内存中的表现形式。

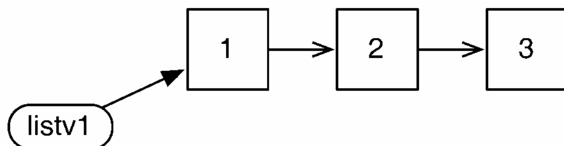


图4-1　`listv1`在内存中的表现形式

现在用cons创建上述列表的修改版，cons返回列表的副本并在副本的首段上添加一个元素：

```
user=> (def listv2 (cons 4 listv1))
#'user/listv2
user=> listv2
(4 1 2 3)
```

新列表可以完全共享原列表的结构——不需要进行复制，如图4-2所示。

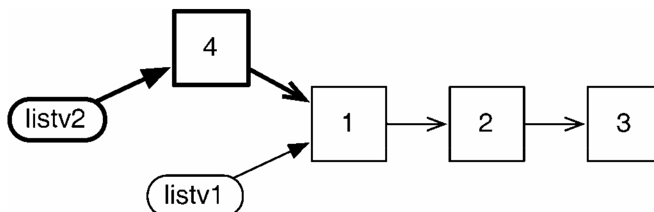


图4-2 listv2在内存中的表现形式

再尝试创建另一个改进版，如图4-3所示。

```
user=> (def listv3 (cons 5 (rest listv1)))
#'user/listv3
user=> listv3
(5 2 3)
```

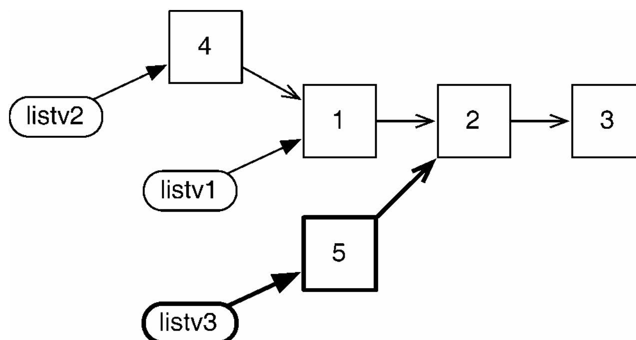


图4-3 listv3在内存中的表现形式

本例中新列表仅共享了原列表的部分结构，但仍不需要进行复制。

有些情况下是不能避免复制的。有共同尾端的列表可以共享结构——如果两个列表具有不同的尾端，就只能进行复制了。举例说明：

```
user=> (def listv1 (list 1 2 3 4))
#'user/listv1
user=> (def listv2 (take 2 listv1))
#'user/listv2
user=> listv2
(1 2)
```

图4-4展示了其在内存中的形式。

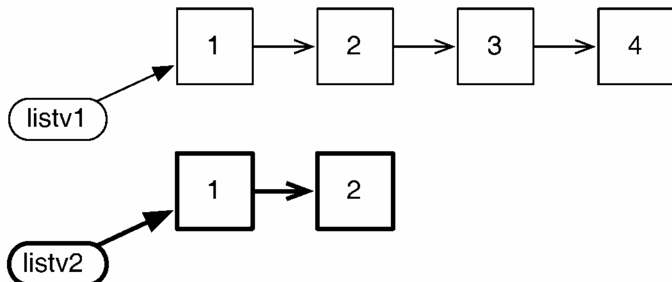


图4-4　具有不同尾端的两个列表在内存中的表现形式

Clojure的集合都是持久的。持久的vector、map和set在实现上都比列表复杂，但此处我们仅需知道它们都使用了共享结构，并且与Ruby和Java中对应的非持久结构具有相近的性能。



小乔爱问：

非函数式语言中数据结构可以是持久的吗？

在非函数式语言中是有可能创建持久数据结构的。我们已经在 Java 中看到过一个例子(`CopyOnWriteArrayList`)，而且 Clojure 的核心数据结构大部分都是由 Java 写成的。而 Java 被创造出来时还不存在 Clojure，所以我们说非函数式语言是有可能创建持久数据结构的。

话虽如此，用非函数式语言实现持久数据结构是比较困难的——很难保证其正确性和效率——主要是因为此类编程语言不能为你提供任何辅助，完全要依靠自己实现持久化。

相比之下，函数式的数据结构天生就是持久的。

标识与状态

如果一个线程引用了持久数据结构，那么其他线程对数据结构的修改对该线程就是不可见的。因此持久数据结构对并发编程的意义非比寻常，其分离了标识（identity）与状态（state）。

你的汽车有多少油？现在这一刻可能有一半油。一段时间以后油箱可能几乎空了，再过几分钟（当你加完油后）油箱就满了。“你的汽车有多少油”是一个标识，其状态是一直在改变的，也就是说，实际上它是一系列不同的值——2012-02-23 12:03，值是0.53；2012-02-23 14:30，值是0.12；2012-02-23 14:31，值是1.00。

命令式语言中，一个变量混合了标识与状态——一个标识只能拥有一个值，这让我们很容易忽略一个事实：状态实际上是随时间变化的一系列值。持久数据结构将标识与状态分离开来——

如果获取了一个标识的当前状态，无论将来对这个标识怎样修改，获取的那个状态将不再改变。

赫拉克利特（Heraclitus）是这样描述这个现象的：

> 我们不能两次踏入同一条河流，因为水在不停地流动。

许多编程语言都错误地认为河流是不变的实体；而Clojure则认为河流是一直在改变的。

重试

由于Clojure是函数式语言，其原子变量是无锁的——其内部实现使用了`java.util.concurrent.AtomicReference`包提供的`compareAndSet()`方法。因此使用原子变量的效率很高且不会发生阻塞（当然也不会有死锁的风险）。但这也要求`swap!`必须处理下面这种情况：当`swap!`调用其参数函数（即由参数传入的函数）产生新值、但还未修改原子变量的值时，其他线程就修改了原子变量的值。

4

如果发生了这种情况，`swap!`就需要重试（retry）。`swap!`将放弃从参数函数中返回的值，并用原子变量的新值重新调用参数函数。我们在2.4节中介绍`ConcurrentHashMap`时见过类似的机制。这要求`swap!`的参数函数必须没有副作用——否则，在重试时这些副作用可能会发生多次。

幸运的是，函数式代码天生是没有副作用的。Clojure的函数式天性让我们避免了这个麻烦。

校验器

假设我们需要一个非负值的原子变量。在创建原子变量时可以提供一个校验器（validator）：

```
user=> (def non-negative (atom 0 :validator #(>= % 0)))
#'user/non-negative
user=> (reset! non-negative 42)
42
user=> (reset! non-negative -1)
IllegalStateException Invalid reference state
```

校验器是一个函数，当改变原子变量的值时就会调用它。如果校验器返回`true`，就允许这次修改，否则就放弃这次修改。

校验器在原子变量的值改变生效**之前**被调用。与本节“重试”部分中传给`swap!`的参数函数类似，当`swap!`进行重试时，校验器可能会被调用多次。因此校验器不应有副作用。

监视器

可以为原子变量添加一个监视器：

```
user=> (def a (atom 0))
#'user/a
```

```
user=> (add-watch a :print #(println "Changed from " %3 " to " %4))
#<Atom@542ab4b1: 0>
user=> (swap! a + 2)
Changed from 0 to 2
2
```

添加监视器时需要提供一个键值和一个监视函数。键值用于区分不同的监视器（例如，有多个监视器时，可以通过键值来删除一个指定的监视器）。原子变量的值被改变时会调用监视器。监视器接受四个参数——调用add-watch时指定的键值、原子变量的引用、原子变量的旧值和原子变量的新值。

上例中再次使用了读取器宏#(...)来定义匿名函数，该函数用于打印原子变量的旧值（%3）和新值（%4）。

与校验器不同，监视器是在原子变量的值改变之后才被调用，且无论swap!重试多少次，监视器只会被调用一次。因此，监视器可以具有副作用。注意：监视器被调用时，原子变量的值可能已经再次被改变，因此监视器必须使用参数中提供的新值，而不能通过对原子变量进行解引用来获取新值。

混搭式Web服务

在3.4节中，我们用Clojure创建了纯粹函数式的Web服务。尽管它运行良好，却存在两个明显的限制——仅能处理一个文本数据，并且会持续消耗内存。在此我们会在保留原版本的函数式特性的同时，突破这两个限制。

会话管理

我们将引入会话（session）这个概念，使Web服务支持多个文本数据。每个会话拥有一个唯一的数字标识，通过下面的代码可生成这个标识：

Clojure/TranscriptHandler/src/server/session.clj

```
(def last-session-id (atom 0))
(defn next-session-id []
  (swap! last-session-id inc))
```

这里会用到原子变量last-session-id，创建新会话标识时会将原子变量的值递增。每次调用next-session-id都会得到比之前大1的一个值：

```
server.core=> (in-ns 'server.session)
#<Namespace server.session>
server.session=> (next-session-id)
1
server.session=> (next-session-id)
2
server.session=> (next-session-id)
3
```

还用到了原子变量sessions，它是将会话标识映射到会话的map，利用sessions可以追踪当前活跃的会话。

```
(def sessions (atom {}))

(defn new-session [initial]
  (let [session-id (next-session-id)]
    (swap! sessions assoc session-id initial)
    session-id))

(defn get-session [id]
  (@sessions id))
```

通过调用new-session并传入一个初始值，可以创建一个新的会话。new-session将获取一个新的会话标识，并调用swap!将会话添加到sessions中。用get-session获取会话的过程就是简单地用会话标识进行查找。

会话过期

如果要解决Web服务持续消耗内存的问题，就需要一种机制来删除不再使用的会话。虽然可以显式地进行删除（比如使用一个delete-session函数），但对于一个Web服务来说，不能依赖于客户端清理各自的会话，而是需要实现会话过期的机制。先对之前的代码做一个小改动：

Clojure/TranscriptHandler/src/server/session.clj

```
(def sessions (atom {}))

➤ (defn now []
➤   (System/currentTimeMillis))

(defn new-session [initial]
  (let [session-id (next-session-id)
➤       session (assoc initial :last-referenced (atom (now)))]
    (swap! sessions assoc session-id session)
    session-id))

(defn get-session [id]
  (let [session (@sessions id)]
➤   (reset! (:last-referenced session) (now))
    session))
```

新的辅助函数now会返回当前时间。用new-session创建新会话时，需要为会话添加新属性:last-referenced，这是含有当前时间的原子变量。当调用get-session访问会话时，用reset!来更新这个时间戳。

现在每个会话都有:last-referenced属性。程序会定期检查所有的会话，当某会话超过一定时间没有被访问时，就可以让该会话过期：

Clojure/TranscriptHandler/src/server/session.clj

```
(defn session-expiry-time []
```

```
  (- (now) (* 10 60 1000)))
(defn expired? [session]
  (< @(:last-referenced session) (session-expiry-time)))

(defn sweep-sessions []
  (swap! sessions #(remove-vals % expired?)))
(def session-sweeper
  (schedule {:min (range 0 60 5)} sweep-sessions))
```

在session-sweeper函数中，使用了Schejulure库[1]，这段代码让程序每5分钟调用一次sweep-sessions。sweep-sessions会（使用Useful库[2]提供的remove-vals函数）删除expired?为true的会话，这些会话最后一次被访问是在session-expiry-time毫秒（即10分钟）之前。

组装

现在可以将会话这个功能添加到之前的Web服务中。首先，需要一个创建新会话的函数：

Clojure/TranscriptHandler/src/server/core.clj

```
(defn create-session []
  (let [snippets (repeatedly promise)
        translations (delay (map translate
                                 (strings->sentences (map deref snippets))))]
    (new-session {:snippets snippets :translations translations})))
```

与上一章相似，我们仍然使用一个无穷的懒惰的promise序列（snippets）来表示接收到的片段，以及一个map（translations）来表示该序列到翻译结果的映射，但这次将两个变量都保存到了会话中。

接下来，修改accept-snippet和get-translation函数，使其从会话中得到:snippets和:translations：

Clojure/TranscriptHandler/src/server/core.clj

```
(defn accept-snippet [session n text]
  (deliver (nth (:snippets session) n) text))

(defn get-translation [session n]
  @(nth @(:translations session) n))
```

最后，需要为路由添加会话功能：

Clojure/TranscriptHandler/src/server/core.clj

```
(defroutes app-routes
  (POST "/session/create" []
    (response (str (create-session))))
```

---

① https://github.com/AdamClements/schejulure

② https://github.com/flatland/useful

```
(context "/session/:session-id" [session-id]
  (let [session (get-session (edn/read-string session-id))]
    (routes
      (PUT "/snippet/:n" [n :as {:keys [body]}]
        (accept-snippet session (edn/read-string n) (slurp body))
        (response "OK"))

      (GET "/translation/:n" [n]
        (response (get-translation session (edn/read-string n))))))))
```

至此，已经得到了新的Web服务，其主体仍是函数式的，并且谨慎地使用了可变状态。

第一天总结

我们结束了第一天的学习。第二天将学习另一些可变数据类型——代理（agent）和引用（ref）。

第一天我们学到了什么

Clojure是一门不纯粹的函数式语言，提供了大量的可变数据类型。我们已经学习了其中最简单的一种——原子变量。

- 命令式语言和不纯粹的函数式语言的区别是今天的一个重点。
  - 命令式语言中，变量默认是状态易变的，代码会经常修改变量。
  - 不纯粹的函数式语言中，变量默认是状态不易变的，代码仅在必要时修改变量。
- 函数式语言中，数据结构是持久的，也就是说当一个线程修改它时，将不会影响到引用同一个数据结构的其他线程。
- 借助上述特性，我们可以分离标识与状态。与标识不同，状态实际上是一系列随时间变化的值。

第一天自习

查找

- 阅读Karl Krukow的博文“Understanding Clojure’s PersistentVector Implementation”，了解比链表更复杂的持久数据结构是如何实现的。
- 阅读上一篇博文的后续文章，了解`PersistentHashMap`是如何通过Hash Array Mapped Trie技术来实现的。

实践

- 扩展本节开头的例子`TournamentServer`，使其可以添加和删除运动员。
- 扩展“混搭式Web服务”中的例子`TranscriptServer`，当一个片段花费10余秒仍未接收完时，让服务器能从这个状态中恢复过来。

4.3　第二天：代理和软件事务内存

我们昨天学习了原子变量，今天来学习其他两种可变数据类型：代理（agent）和引用（ref）。与原子变量性质相同，代理和引用都可以用于并发，也能与持久数据结构一起使用，实现标识与状态的分离。在学习引用时，我们将讨论Clojure如何实现对软件事务内存的支持，使变量在无锁的情况下可以被并行地修改，同时仍保持一致性。

代理

与原子变量类似，代理包含了对一个值的引用，可以通过deref或@获取该值：

```
user=> (def my-agent (agent 0))
#'user/my-agent
user=> @my-agent
0
```

调用send函数可以修改代理的值：

```
user=> (send my-agent inc)
#<Agent@2cadd45e: 1>
user=> @my-agent
1
user=> (send my-agent + 2)
#<Agent@2cadd45e: 1>
user=> @my-agent
3
```

与swap!类似，send接受一个函数（可以附加额外的参数），并用代理的当前值作为参数对该函数进行调用，函数的返回值将作为代理的新值。

send与swap!的区别是，前者会（在代理的值更新之前）立刻返回——传给send的函数将在之后的某个时间被调用。如果多个线程同时调用send，传给send的函数将被串行调用：同一时间只会调用一个。也就是说该函数不会进行重试，并且可以具有副作用。



小乔爱问：

代理是actor吗？

表面看上去 Clojure 的代理和 actor（将在第 5 章介绍）非常相似。但这是一种误解，实际上两者有很大差异：

- 通过deref可以获得代理的值，而actor没有提供直接获得值的方式；
- actor可以包含行为（behavior），而代理则不可以——对数据的操作函数必须由调用者提供；
- actor提供了复杂的错误检测和错误恢复的机制，而代理仅提供了简单的错误报告机制；
- actor能支持分布式，而代理则不能；
- 使用多个actor可能会引发死锁，而使用多个代理则不会。

等待代理的操作完成

之前我们在REPL的输出中看到，send的返回值是一个代理的引用。REPL打印这个引用时，也打印了代理的值——本例中是1：

```
user=> (send my-agent inc)
#<Agent@2cadd45e: 1>
```

再次调用send时，其显示的不是3，而仍然是1：

```
user=> (send my-agent + 2)
#<Agent@2cadd45e: 1>
```

这是因为传给send的函数是异步运行的，在REPL获得代理的值之前，该函数可能已经运行，也可能没有运行。对于执行比较快的函数，在REPL获得代理的值之前可能已经运行了；但如果我们用Thread/sleep来延长函数的运行时间，那函数就不大可能在REPL获取代理的值之前完成运行：

```
user=> (def my-agent (agent 0))
#'user/my-agent
user=> (send my-agent #((Thread/sleep 2000) (inc %)))
#<Agent@224e59d9: 0>
user=> @my-agent
0
user=> @my-agent
1
```


Clojure提供了await函数，这个函数将一直阻塞，直到由当前线程派给某个（或某些）代理的所有操作全部完成（Clojure还提供了await-for函数，可以指定等待的超时时间）：

```
user=> (def my-agent (agent 0))
#'user/my-agent
user=> (send my-agent #((Thread/sleep 2000) (inc %)))
#<Agent@7f5ff9d0: 0>
user=> (await my-agent)
nil
user=> @my-agent
1
```



小乔爱问：

Send-Off和Send-Via

除了send，代理还支持send-off和send-via函数。不同的是如何执行传入的函数，send使用公用线程池，send-off使用一个新线程，而send-via使用由参数指定的executor。

如果传入的函数可能会阻塞（并占用其执行线程）或需要执行很久，推荐使用send-off或send-via。除此之外，三个函数差别不大。

异步更新比同步更新有着明显的优势，尤其是当更新操作会发生阻塞或需要执行很久时。但异步更新也更复杂，尤其在错误处理方面。下面来看看Clojure对错误处理方面的支持。

错误处理

代理与原子变量一样都支持校验器和监视器。下面的例子中，代理使用了一个校验器来确保其值不为负数：

```
user=> (def non-negative (agent 1 :validator (fn [new-val] (>= new-val 0))))
#'user/non-negative
```

递减代理的值，直到其将变为负数：

```
user=> (send non-negative dec)
#<Agent@6257d812: 0>
user=> @non-negative
0
user=> (send non-negative dec)
#<Agent@6257d812: 0>
user=> @non-negative
0
```

不出所料，代理的值不会变为负数。但如果继续使用这个发生过错误的代理，会怎样呢？

```
user=> (send non-negative inc)
IllegalStateException Invalid reference state clojure.lang.ARef.validate⋯

user=> @non-negative
0
```

一旦代理发生错误，就会默认进入失效状态，之后对代理数据的任何操作都会失败。使用`agent-error`可以查看一个代理是否为失效状态（也可以查看其失效原因），使用`restart-agent`可以重置失效状态的代理：

```
user=> (agent-error non-negative)
#<IllegalStateException java.lang.IllegalStateException: Invalid reference state>
user=> (restart-agent non-negative 0)
0
user=> (agent-error non-negative)
nil
user=> (send non-negative inc)
#<Agent@6257d812: 1>
user=> @non-negative
1
```

创建代理时其错误处理模式默认为`:fail`。也可以将错误处理模式置为`:continue`，这意味着失效状态的代理不需要通过`restart-agent`重置就可以处理新的操作。如果设置了错误处理函数，那错误处理模式默认为`:continue`，代理出现错误时则会调用错误处理函数。

下面来看一个使用代理的例子。

内存日志系统

进行并行编程时，我发现内存日志系统是非常有用的——传统日志系统体量过大，在处理并发问题时往往无所裨益，比如对于每行日志都会进行多次上下文切换和IO操作。用线程与锁模型实现内存日志系统比较复杂，而使用代理来实现就非常简单：

Clojure/Logger/src/logger/core.clj

```
(def log-entries (agent []))

(defn log [entry]
  (send log-entries conj [(now) entry]))
```

日志被记录在代理`log-entries`中，其初始值是一个空数组。`log`函数用`conj`向数组中添加新元素，新元素是一个二元数组——第一个元素是时间戳（记录着`send`被调用的时间，而不是`conj`被代理调用的时间，`conj`调用的时间可能比`send`要晚），第二个元素是日志消息。

在REPL中验证一下：

```
logger.core=> (log "Something happened")
#<Agent@bd99597: [[1366822537794 "Something happened"]]>
logger.core=> (log "Something else happened")
#<Agent@bd99597: [[1366822538932 "Something happened"]]>
logger.core=> @log-entries
[[1366822537794 "Something happened"] [1366822538932 "Something else happened"]]
```

下一节我们来学习另一种Clojure的共享可变数据类型——引用。

软件事务内存

引用（ref）比原子变量和代理更复杂，通过引用可以实现软件事务内存（Software Transactional Memory，STM）。通过原子变量和代理每次仅能修改一个变量，而通过STM可以对多个变量进行并发的一致的修改，就像数据库的事务可以对多条记录进行并发的一致的修改一样。

与原子变量和代理类似，引用（ref）包装了对一个值的引用（reference），可以通过`deref`或@获取该值。

```
user=> (def my-ref (ref 0))
#'user/my-ref
user=> @my-ref
0
```

引用的值可以通过`ref-set`来设置。Clojure提供了`alter`函数来修改引用的值，它类似于之前提到的`swap!`和`send`，但不同点在于其使用时不能只是简单地被调用：

```
user=> (ref-set my-ref 42)
IllegalStateException No transaction running

user=> (alter my-ref inc)
```

4

```
IllegalStateException No transaction running
```

只能在一个事务中才能修改引用的值。

事务

STM事务具有原子性、一致性和隔离性。

原子性：在其他的事务看来，当前事务的所有副作用或者全部发生，或者都不发生。

一致性：事务保证全程遵守校验器定义的规范（就像我们在原子变量和代理中看到的一样）。如果事务的一系列修改中任一个校验失败，那么所有的修改都不会发生。

隔离性：多个事务可以同时运行，但同时运行的事务的结果与串行运行这些事务的结果应当完全一样。

你可能已经看出来了，这三个性质是许多数据库支持的ACID特性中的前三个。唯一遗漏的性质是持久性——STM的数据在电源故障或系统崩溃时会丢失。如果需要用到持久性，就必须使用数据库。

使用dosync创建一个事务：

```
user=> (dosync (ref-set my-ref 42))
42
user=> @my-ref
42
user=> (dosync (alter my-ref inc))
43
user=> @my-ref
43
```

dosync包装的所有元素构成了一个事务。



小乔爱问：

这些事务必须具有隔离性吗?

大多数场景适合使用完全隔离的事务，但对于有些场景，隔离性是个过强的约束。如果用 commute 替换 alter，就可以得到不那么强的隔离性。

虽然使用 commute 是一种有效的优化手段，但是理解其适用场景是比较复杂的，因此本书不介绍这部分内容。

多个引用

事务通常会涉及多个引用（否则，应使用原子变量或代理）。使用事务的典型场景是在不同银行账户之间进行转账——大家永远不想看到“钱已经从源账户中划出、但未能划入目标账户”的情况。下面这个函数保证了出账和入账都发生，或者都不发生：

Clojure/Transfer/src/transfer/core.clj

```
(defn transfer [from to amount]
  (dosync
    (alter from - amount)
    (alter to + amount)))
```

对这个函数进行验证：

```
user=> (def checking (ref 1000))
#'user/checking
user=> (def savings (ref 2000))
#'user/savings
user=> (transfer savings checking 100)
1100
user=> @checking
1100
user=> @savings
1900
```

如果STM运行时检测到几个并发事务的修改发生冲突，那其中的一个或几个事务将进行重试。就像修改原子变量一样，事务需要保证没有副作用。

重试事务

秉着先做实验再讲理论的精神，我们先对transfer函数进行压力测试，看看是否能找到事务被重试的现象。尝试以下代码：

Clojure/Transfer/src/transfer/core.clj

```
(def attempts (atom 0))
(def transfers (agent 0))

(defn transfer [from to amount]
  (dosync
➤   (swap! attempts inc) // 在事务内产生副作用——产品代码中永远不要这样写!!!
➤   (send transfers inc)
    (alter from - amount)
    (alter to + amount)))
```

这段代码故意打破“禁止副作用”的规则，在事务中修改原子变量来产生副作用。我们是为了做事务重现的实验才这样做，永远不要在产品代码中这样写。

除了在原子变量中维护了计数器，我们还在代理中维护了计数器，稍后会对这个做法进行解释。

下面这段代码用于对transfer函数进行压力测试：

Clojure/Transfer/src/transfer/core.clj

```
(def checking (ref 10000))
(def savings (ref 20000))

(defn stress-thread [from to iterations amount]
  (Thread. #(dotimes [_ iterations] (transfer from to amount))))

(defn -main [& args]
  (println "Before: Checking =" @checking " Savings =" @savings)
  (let [t1 (stress-thread checking savings 100 100)
        t2 (stress-thread savings checking 200 100)]
    (.start t1)
    (.start t2)
    (.join t1)
    (.join t2))
  (await transfers)
  (println "Attempts: " @attempts)
  (println "Transfers: " @transfers)
  (println "After: Checking =" @checking " Savings =" @savings))
```

这段代码创建了两个线程。一个线程从支票账户往储蓄账户进行100次转账，每次100美金；另一个线程从储蓄账户往支票账户进行200次转账，每次100美金。我运行时得到以下输出：

```
Before: Checking = 10000 Savings = 20000
Attempts: 638
Transfers: 300
After: Checking = 20000 Savings = 10000
```

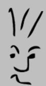
太好了，我们得到了预期的结果，STM运行时确保并发事务运行得到了正确的结果。其代价是进行了多次重试（本例中进行了338次重试），而好处是全程没有使用锁，不会有发生死锁的风险。

当然，这并不是现实中的情况——两个线程在频繁的循环中访问同一个引用，在此情况下会不断发生访问冲突。实际情况中，一个设计良好的系统的事务重试次数会少得多。

事务的安全副作用

你也许注意到尽管原子变量维护的计数不断增大，但代理维护的计数却与事务的数量相等。其原因是代理具有事务性。

如果在事务中用send来更新一个代理，那send仅在事务成功时生效。如果需要在事务成功时产生一些副作用，那send将是最佳选择。



小乔爱问：

函数名末尾的感叹号是什么意思？

你也许注意到一些函数名末尾有个感叹号——这种命名规则在表达什么？

Clojure 用一个感叹号表示一个函数不是事务安全的，比如`swap!`和`reset!`。由于更新代理的函数使用的是`send`而不是`send!`，所以可以安全地在事务中更新代理。

Clojure对共享可变状态的支持

之前已经学习了Clojure支持共享可变状态的三种机制。每种机制都有各自的适用场景。

原子变量可以对一个值进行同步更新——同步的意思是更新在`swap!`返回时已经完成。对多个原子变量不能进行一致地更新。

代理可以对一个值进行异步更新——异步的意思是更新可能在`send`返回后进行。对多个代理不能一致更新。

引用可以对多个值进行一致的、同步的更新。

第二天总结

我们完成了第二天的学习。第三天我们将学习一些使用可变类型的复杂例子，并学习不同可变类型的适用场景。

第二天我们学到了什么

除了原子变量，Clojure还提供了代理和引用。

- 原子变量可以对单一值进行隔离的、同步的更新。
- 代理可以对单一值进行隔离的、异步的更新。
- 引用可以对多个值进行一致的、同步的更新。

第二天自习

查找

- 观看Rich Hickey的演讲 “Persistent Data Structures and Managed References: Clojure’s Approach to Identity and State”。
- 观看Rich Hickey的演讲 “Simple Made Easy”。

实践

- 改进4.2节的例子TournamentServer，用引用和事务来实现一个运行井字棋游戏的服务器。
- 用列表存储节点，实现一个持久化的查询二叉树。最坏情况下需要进行多少次复制？平均情况下呢？
- 学习finger tree，并用它实现查询二叉树。它对平均情况下和最坏情况下的性能有什么影响？

4.4　第三天：深入学习

我们已经学习了“Clojure之道”涉及的所有组件。今天将学习一些运用这些组件的复杂例子，以及在面对某个并发问题时应当选择原子变量还是选择STM。

用STM解决哲学家进餐问题

作为开场，先来回顾一下第2章提到的“哲学家进餐问题”，并用Clojure的STM来解决这个问题。该解决方案非常类似于2.3节中介绍的使用条件变量的方案（然而STM方案会更简单）。

我们使用一个引用来代表一个哲学家，引用的值是哲学家的当前状态（:thinking或:eating）。这些引用保存在数组philosophers中。

Clojure/DiningPhilosphersSTM/src/philosophers/core.clj

```
(def philosophers (into [] (repeatedly 5 #(ref :thinking))))
```

每个哲学家都有一个对应的线程：

Clojure/DiningPhilosphersSTM/src/philosophers/core.clj

```
Line 1 (defn think []
     -   (Thread/sleep (rand 1000)))
     -
     - (defn eat []
     5   (Thread/sleep (rand 1000)))
     -
     - (defn philosopher-thread [n]
     -   (Thread.
     -     #(let [philosopher (philosophers n)
    10            left (philosophers (mod (- n 1) 5))
     -            right (philosophers (mod (+ n 1) 5))]
     -        (while true
     -          (think)
     -          (when (claim-chopsticks philosopher left right)
    15            (eat)
     -            (release-chopsticks philosopher))))))
     -
```

```
- (defn -main [& args]
-   (let [threads (map philosopher-thread (range 5))]
20    (doseq [thread threads] (.start thread))
-     (doseq [thread threads] (.join thread))))
```

与Java版本的方案类似，每个线程将无限循环下去（第12行），哲学家或者进行思考，或者尝试进餐。如果claim-chopsticks执行成功（第14行），控制结构when会先调用eat再调用release-chopsticks。

release-chopsticks的实现非常简单：

Clojure/DiningPhilosphersSTM/src/philosophers/core.clj

```
(defn release-chopsticks [philosopher]
  (dosync (ref-set philosopher :thinking)))
```

这段代码仅简单地用dosync创建一个事务，并用ref-set将状态置为:thinking。

首次尝试

claim-chopsticks函数非常值得讨论——先尝试一种实现方法：

```
(defn claim-chopsticks [philosopher left right]
  (dosync
    (when (and (= @left :thinking) (= @right :thinking))
      (ref-set philosopher :eating))))
```

类似于release-chopsticks，这段代码首先创建了一个事务。在这个事务中，检查左边和右边的哲学家的状态——如果两边的状态都是:thinking，就使用ref-set将当前哲学家的状态置为eating。如果条件不满足，when语句将返回nil，即当哲学家无法获得两支筷子并开始进餐时，claim-chopsticks也将返回nil。

尝试运行这段代码，第一感觉是一切运行正常。但偶尔也会发现两个相邻的哲学家在同时进餐，他们共用了一支筷子，显然这是个错误的状态。到底发生了什么？

造成这个问题的原因是我们用@访问了left和right的值。Clojure的STM会保证两个事务不能对同一个引用进行不一致的修改，虽然这段代码并没有修改left或者right，但却读取了它们的值。其他事务是可以修改这些值的，这也就造成了相邻的哲学家同时进餐的错误状态。

确保值不被修改

要解决上面的问题，可以用ensure代替@来访问left和right：

Clojure/DiningPhilosphersSTM/src/philosophers/core.clj

```
(defn claim-chopsticks [philosopher left right]
  (dosync
    (when (and (= (ensure left) :thinking) (= (ensure right) :thinking))
      (ref-set philosopher :eating))))
```

ensure函数确保了其返回的某引用的值不会被其他事务修改。与之前基于线程与锁的解决方案相比较会发现，现在的方案不仅非常简洁，而且是无锁的，即没有死锁风险。

现在的解决方案使用了多个引用和事务，下一节将介绍另一种解决方案，其只使用了一个原子变量。

不用STM解决哲学家进餐问题

面对哲学家进餐问题，除了基于STM的方案我们还有其他选择。之前的方案将每个哲学家都用一个引用来代表，并使用事务来确保对多个引用的更新是一致的。首先，将仅用一个原子变量来表示所有哲学家的状态：

Clojure/DiningPhilosphersAtom/src/philosophers/core.clj

```
(def philosophers (atom (into [] (repeat 5 :thinking))))
```

原子变量的值是一个状态数组。举例说明，哲学家0和哲学家3正在进餐时，其状态数组是：

```
[:eating :thinking :thinking :eating :thinking]
```

然后，要用状态数组中的序号来代表哲学家，就要对philosopher-thread做一点小修改：

Clojure/DiningPhilosphersAtom/src/philosophers/core.clj

```
(defn philosopher-thread [philosopher]
  (Thread.
➤   #(let [left (mod (- philosopher 1) 5)
➤          right (mod (+ philosopher 1) 5)]
      (while true
        (think)
        (when (claim-chopsticks! philosopher left right)
          (eat)
          (release-chopsticks! philosopher))))))
```

之后，要实现release-chopsticks!，只需要用swap!将状态数组的相关元素置为:thinking：

Clojure/DiningPhilosphersAtom/src/philosophers/core.clj

```
(defn release-chopsticks! [philosopher]
  (swap! philosophers assoc philosopher :thinking))
```

这里用到了assoc，之前我们对map用过这个函数，其对数组的效果是类似的：

```
user=> (assoc [:a :a :a :a] 2 :b)
[:a :a :b :a]
```

最后，实现最关键的函数claim-chopsticks!：

Clojure/DiningPhilosphersAtom/src/philosophers/core.clj

```
(defn claim-chopsticks! [philosopher left right]
```

```
(swap! philosophers
  (fn [ps]
    (if (and (= (ps left) :thinking) (= (ps right) :thinking))
      (assoc ps philosopher :eating)
      ps)))
(= (@philosophers philosopher) :eating))
```

传给swap!的匿名函数的参数是状态数组philosophers的当前值，该匿名函数对邻座哲学家的状态进行检查。如果邻座都在思考，就用assoc将当前哲学家的状态置为:eating，否则直接返回philosophers而不改变philosophers的值。

claim-chopsticks!的最后一行代码检查了philosophers的新值，目的是判断swap!是否成功地将当前哲学家的状态置为:eating。[①]

我们已经学习了两种哲学家进餐问题的解决方案，一种使用STM，另一种则不使用。如何在两者之间进行选择呢？

原子变量还是STM？

第二天已介绍过，原子变量可以对单一值进行更新，而引用可以对多个值进行一致的更新。虽然两者功能不同，但如本章所述，我们能做出一个使用STM并涉及多个引用的解决方案，也能很容易将其转换成一个使用单一原子变量的方案。

现在我们面临一个尴尬的局面——当解决一个涉及多个值需一致更新的问题时，即可以使用多个引用并通过事务来保证访问一致性，也可以将这些值整合到一个数据结构中并用一个原子变量管理这个数据结构的访问一致性。

应该如何选择呢？

这个问题的答案很大程度上因人而异——两种方案都正确，所以要选择那个最简便的。在性能上，根据使用场景的特点和数据访问模式的不同，肯定会有所差异，所以需要用压力测试进行性能评估之后再进行选择。

虽然STM带有更多光芒，但有经验的Clojure程序员知道：由于语言的函数性减少了对可变量的使用，因此大部分问题都可以用原子变量来解决。更简单的方案通常会更有效。

定制并发函数

使用原子变量解决哲学家进餐问题的代码是可以正确运行的，但claim-chopsticks!的实

① 此处作者在swap!之后调用了@philosophers，设想如果在swap!之后、@philosophers之前，另一个线程修改了当前哲学家的状态，那么claim-chopsticks!的返回值将不正确。所以不推荐这样使用，而是应将状态设置和状态检查放在同一个原子操作中。不过，此处由于当前哲学家状态仅能由当前线程修改，所以这段代码是正确的。
——译者注

现并不优雅。如果当前哲学家可以获得两边的筷子，那调用swap!之后的那次检查是否必要？理想状况下，仅需要一个类似于swap!的函数，其接受一个参数作为判断条件，仅当判断条件为真时进行swap!操作。可以将claim-chopsticks!写成这样：

Clojure/DiningPhilosphersAtom2/src/philosophers/core.clj

```
(defn claim-chopsticks! [philosopher left right]
  (swap-when! Philosophers
    #(and (= (%1 left) :thinking) (= (%1 right) :thinking))
    assoc philosopher :eating))
```

Clojure并没有提供我们理想中的函数，但我们可以自己写一个：

Clojure/DiningPhilosphersAtom2/src/philosophers/util.clj

```
Line 1 (defn swap-when!
     -   "If (pred current-value-of-atom) is true, atomically swaps the value
     -   of the atom to become (apply f current-value-of-atom args). Note that
     -   both pred and f may be called multiple times and thus should be free
     5   of side effects. Returns the value that was swapped in if the
     -   predicate was true, nil otherwise."
     -   [a pred f & args]
     -   (loop []
     -     (let [old @a]
    10       (if (pred old)
     -         (let [new (apply f old args)]
     -           (if (compare-and-set! a old new)
     -             new
     -             (recur)))
    15         nil))))
```

这段代码用到了一些先前没出现过的语言特性。第一个特性是文档字符串。文档字符串是位于defn和参数列表之间的字符串，用于描述函数的行为。文档字符串对任何函数都是有益的，尤其是对那些为了重用而设计的辅助函数。除了在源码中查看文档字符串，也可以从REPL中获取：

```
philosophers.core=> (require '[philosophers.util :refer :all])
nil
philosophers.core=> (clojure.repl/doc swap-when!)
-------------------------
philosophers.util/swap-when!
([atom pred f & args])
  If (pred current-value-of-atom) is true, atomically swaps the value
  of the atom to become (apply f current-value-of-atom args). Note that
  both pred and f may be called multiple times and thus should be free
  of side effects. Returns the value that was swapped in if the
  predicate was true, nil otherwise.
```

第二个特性是参数列表中的&符号，其说明了swap-when!的参数个数是可变的（非常类似于Java中的省略号或Ruby中的星号）。通过名为args的数组可以访问附加的参数。这里还用到了apply，其可以将最后一个参数展开，作为附加的参数传给f（第11行）——举例说明，下面这两种调用+函数的方式是等价的：

```
user=> (apply + 1 2 [3 4 5])
15
user=> (+ 1 2 3 4 5)
15
```

我们还使用较底层的compare-and-set!（第12行）来替换swap!。compare-and-set!接受一个原子变量、原子变量的旧值和原子变量的新值——仅当原子变量的当前值等于旧值时，原子变量的值会被更新为新值，整个比较和更新的过程都是原子的。

当compare-and-set!成功时，swap-when!返回原子变量的新值。否则，使用recur（第14行）回到第8行重新运行。



小乔爱问：

什么是Loop/Recur?

与许多函数式语言不同，Clojure 不具备尾调用消除（tail-call elimination）的能力，因此 Clojure 代码不常使用递归，而是用 loop/recur。

loop 宏定义了一个锚点，recur 可以跳到这个锚点（类似于 C/C++中的 setjmp()和 longjmp()）。Clojure 语言手册中详述了其工作原理。

第三天总结

我们完成了第三天的学习，讨论了Clojure如何将函数式编程与可变量混搭使用。

第三天我们学到了什么

Clojure的函数式性质极大地减少了可变量的使用。通常情况下，基于原子变量的简单并发方案[①]就足够了：

- 基于STM的解决方案（通过事务来达成多个值的一致性）可以被基于代理的解决方案（将多个值整合到一个数据结构中，并用一个代理来管理对这个数据结构的访问）替换；
- 在两种方案之间的选择往往基于个人风格和程序性能；
- 定制并发函数可以让代码更简洁。

第三天自习

查找

① 可以认为基于原子变量的方案是基于代理的方案的子集。——译者注

- 观看Rich Hickey的演讲“The Database as a Value”，注意Datomic[①]是如何有效地将整个数据库视为一个单一值的。

实践

- 改进第二天的实践部分的`TournamentServer`，用原子变量代替引用和事务。哪一种方案更简单？哪一份代码更易读？哪一种方案性能更好？

4.5　复习

Clojure在解决并发问题上非常务实。起先我们意识到并发编程中最大的障碍来自于共享可变状态，于是Clojure作为一门函数式语言挺身而出，编写出具有引用透明性且无副作用的代码。之后我们又意识到一些问题场景需要对某些可变状态进行维护，因此Clojure又提供了很多并发安全的可变数据类型。

优点

很显然，本章“Clojure之道”的优点建立在上一章介绍的函数式编程的基础上。我们可以用Clojure“函数式地”解决函数式的问题，也可以在必要的时候突破函数式的禁锢。

传统命令式语言的变量混淆了标识与状态这两个概念，而Clojure的持久数据结构将可变量的标识与状态分离开来。这解决了使用锁的方案的大部分缺点。专家级Clojure程序员知道解决并发问题的最佳选择是那个“刚刚够用”的方案。

缺点

“Clojure之道”的主要缺点在于不支持分布式（地理分布或其他）编程。与之相关，它也无法直接提供容错性。

由于Clojure在JVM中运行，很多第三方库可以为Clojure弥补这些缺点（Akka[②]就是其中之一，它使用了下一章将要介绍的actor模型），不过对这些第三方库的介绍超出了本书的范围。

其他语言

Haskell提供了类似本章介绍的功能，不过作为一种纯粹的函数式语言，使用起来会有一种不同的“体验”。值得一提的是Haskell提供了完整的STM实现，Simon Peyton Jones的文章“Beautiful

---

① http://www.datomic.com

② http://blog.darevay.com/2011/06/clojure-and-akka-a-match-made-in/

Concurrency”[1]对其进行了详细的解说。

另外，大部分主流编程语言都提供了STM的实现，包括GCC支持的编程语言[2]。尽管如此，有证据表明：STM模型并不适合于命令式语言[3]。

结语

Clojure在函数式编程和可变状态之间取得了很好的平衡，比起纯粹的函数式语言，这些命令式语言的特性会让程序员感觉更亲切、更容易上手。与此同时Clojure保留了函数式编程的大部分优点，包括对并发的完美支持。

概括而言，Clojure精心设计了用于并发的语义，从而保留了共享可变状态。下一章我们将学习actor模型，它完全抛弃了共享可变状态。

① http://research.microsoft.com/pubs/74063/beautiful.pdf
② http://gcc.gnu.org/wiki/TransactionalMemory
③ http://www.infoq.com/news/2010/05/STM-Dropped

第 5 章

Actor

使用actor就像租车——我们如果需要，可以快速便捷地租到一辆；如果车辆发生故障，也不需要自己修理，直接打电话给租车公司更换另外一辆即可。

actor模型是一种适用性非常好的通用并发编程模型。它可以应用于共享内存架构和分布式内存架构，适合解决地理分布型的问题。同时它还能提供很好的容错性。

5.1 更加面向对象

函数式编程不使用可变状态，也就避免了共享可变状态带来的问题。相比之下，使用actor模型保留了可变状态，只是不进行共享。

actor类似于面向对象（OO）编程中的对象——其封装了状态，并通过消息与其他actor通信。两者的区别是所有actor可以同时运行，而且，与OO式的“消息传递”（实质上只是调用一个方法）不同，actor之间的消息传递是*真实地*在传递消息。

actor模型是一个通用的并发编程模型，几乎可以用在任何一种编程语言里，最典型的是Erlang[①]。而我们将用Elixir[②]来介绍actor模型，它是Erlang虚拟机（BEAM）上相对较新的一门编程语言。

与Clojure类似，Elixir是一门不纯粹的、动态类型的函数式语言。如果你熟悉Java或者Ruby，很容易就能看懂Elixir代码。与以往一样，我们不会把本章写成Elixir的教程（本书的主旨是并发，而不是编程语言），但仍将介绍一些必要的语言特性。如果你对这门语言并不熟悉，那就不得不在某些地方“盲目”接受本书的说法——如果想深入学习Elixir，推荐阅读*Programming Elixir* [Tho14]。

第一天，我们将学习actor模型的基础——如何创建actor、发送消息和接收消息。第二天，学习使用actor模型的程序具有容错性的关键：失败检测和“任其崩溃”的哲学。第三天，学习如何

① http://www.erlang.org/

② http://elixir-lang.org/

通过actor模型编写分布式程序，将计算扩展到多台计算机，并能从一台或多台计算机的崩溃中恢复过来。

5.2 第一天：消息和信箱

现在我们来学习如何创建和停止进程、如何发送和接收消息，以及如何检测进程已终止。



小乔爱问：

是actor还是进程？

类似于 Erlang，在 Elixir 中，actor 对象被称为进程。大部分场景下，进程是一个重量级的概念，它会消耗很多资源，且创建代价很高。不过在 Elixir 中，进程是一个轻量级的概念，比操作系统级的线程还要轻量：它消耗更少的资源，且创建代价很低。Elixir 程序可以毫无困难地创建数千个进程，通常不需要依赖线程池（参见 2.4 节）等技术。

第一个actor

先来尝试创建一个简单的actor，并向其发送一些消息。我们将创建一个叫Talker的actor，其收到不同的消息时会输出不同的结果。

所发送的消息是一个元组（tuple）——元组是一个由多个值组成的序列。在Elixir中，用花括号（{}）表示元组，举例如下：

```
{:foo, "this", 42}
```

这是个三元组，第一个元素是一个关键字（与Clojure类似，也是用冒号表示关键字），第二个元素是一个字符串，第三个元素是一个整数。

来看看actor的代码：

Actors/hello_actors/hello_actors.exs

```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
    end
    loop
  end
end
```

我们稍后会对这段代码进行深入分析。现在只需要知道这段代码定义了一个actor，其接受三种不同的消息，并打印不同的字符串。

下面创建这个actor的实例，并向其发送一些消息：

Actors/hello_actors/hello_actors.exs

```
pid = spawn(&Talker.loop/0)
send(pid, {:greet, "Huey"})
send(pid, {:praise, "Dewey"})
send(pid, {:celebrate, "Louie", 16})
sleep(1000)
```

首先，这段代码用`spawn`创建了actor的实例，并获得进程标识符，这个进程标识符被保存在`pid`中。进程将执行`spawn`的参数所指定的函数，本例中的函数是`Talker`模块中的`loop()`，该函数接受0个参数。

然后，这段代码向刚创建的actor实例发送了三个消息。

最后，sleep一下，让各个进程有时间处理消息（用`sleep()`并不是最佳选择——稍后将介绍如何改进）。

以下是这段代码的运行结果：

```
Hello Huey
Dewey, you're amazing
Here's to another 16 years, Louie
```

我们已经学习了如何创建actor并向其发送消息，现在需要剖析一下其中的机制。

队列式信箱

异步地发送消息是用actor模型编程的重要特性之一。消息并不是直接发送到一个actor，而是发送到一个信箱（mailbox），如图5-1所示。

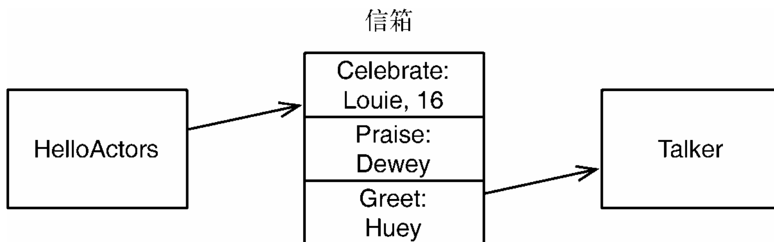


图5-1　向信箱发送消息

这样的设计解耦了actor之间的关系——actor都以自己的步调运行，且发送消息时不会被阻塞。

虽然所有actor可以同时运行，但它们都按照信箱接收消息的顺序来依次处理消息，且仅在当

前消息处理完成后才会处理下一个消息，因此我们只需要关心发送消息时的并发问题即可。

接收消息

通常actor会进行无限循环，通过receive等待接收消息，并进行消息处理。现在来看一下Talker的循环代码：

Actors/hello_actors/hello_actors.exs

```
def loop do
  receive do
    {:greet, name} -> IO.puts("Hello #{name}")
    {:praise, name} -> IO.puts("#{name}, you're amazing")
    {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
  end
  loop
end
```

该函数通过递归调用自己来进行无限循环，用receive块来等待一个消息，通过匹配模式来决定如何处理消息。这段代码依次用每个模式对接收到的消息进行匹配——一旦匹配成功，在箭头（->）右边的代码中，就可以通过模式中的变量（name和age）来访问消息中的对应值。处理消息的代码使用字符串插值技术来构造字符串并输出——字符串插值技术指的是#{...}中的代码将被求值并将求值结果插入到字符串的对应位置。

5

"第一个actor"部分的最后一段代码在退出前sleep了1秒，这样才有足够的时间处理消息。这个解决方案不过是差强人意——我们可以做得更好。



小乔爱问：

不断递归难道不会栈溢出吗?

你也许已经注意到了 Talker 的 loop()函数不断地进行递归，就会担心堆栈会不断被消耗。幸运的是我们并不需要担心——与许多函数式语言一样（不过 Clojure 是个特例，参见 4.4 节的"小乔爱问"），Elixir 实现了尾调用消除。尾调用消除指的是如果函数在最后调用了自己，那么递归调用将被替换成一个简单的跳转。

连接到（linking）进程

为了彻底关闭一个actor，需要满足两个条件。第一个是需要告诉actor在完成消息处理后就关闭；第二个是需要知道actor何时完成关闭。

首先，通过接收一个显式的关闭消息（类似于2.4节中介绍的毒丸）来满足第一个条件：

Actors/hello_actors/hello_actors2.exs

```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
➤     {:shutdown} -> exit(:normal)
    end
    loop
  end
end
```

然后，需要一个方法来获知actor是否完全关闭。下述代码将:trap_exit设为true，并用spawn_link()替换spawn()以连接到进程：

Actors/hello_actors/hello_actors2.exs

```
Process.flag(:trap_exit, true)
pid = spawn_link(&Talker.loop/0)
```

现在当创建的进程关闭时，就会得到一个通知（是一个系统产生的消息）。这个消息是一个三元组：

```
{:EXIT, pid, reason}
```

最后，发送关闭消息并收到关闭通知：

Actors/hello_actors/hello_actors2.exs

```
send(pid, {:greet, "Huey"})
send(pid, {:praise, "Dewey"})
send(pid, {:celebrate, "Louie", 16})
➤ send(pid, {:shutdown})

➤ receive do
➤   {:EXIT, ^pid, reason} -> IO.puts("Talker has exited (#{reason})")
➤ end
```

receive模式中使用^符号（脱字符）的第二个元素，将不会绑定到消息的第二个数据，而是用pid的当前值进行模式匹配。

运行这个新版本代码，输出如下：

```
Hello Huey
Dewey, you're amazing
Here's to another 16 years, Louie
Talker has exited (normal)
```

我们将在第二天深入讨论连接技术。

有状态的actor

之前的Talker是没有状态的actor。创建一个有状态的actor时，很容易想到使用可变量，但实际上可以使用递归。举例说明，下面这个actor每收到一个消息时都会将计数器加1：

Actors/counter/counter.ex

```
defmodule Counter do
  def loop(count) do
    receive do
      {:next} ->
        IO.puts("Current count: #{count}")
        loop(count + 1)
    end
  end
end
```

在交互式Elixir环境iex（Elixir版的REPL）中运行这段代码：

```
iex(1)> counter = spawn(Counter, :loop, [1])
#PID<0.47.0>
iex(2)> send(counter, {:next})
Current count: 1
{:next}
iex(3)> send(counter, {:next})
{:next}
Current count: 2
iex(4)> send(counter, {:next})
{:next}
Current count: 3
```

这段代码使用了接受三个参数的spawn()：第一个是模块名，第二个是模块中的函数名，第三个是参数列表。用这个版本的spawn()可以将初始的count传给Counter.loop()。不出所料，每发送一个{:next}消息，这个actor就会输出一个不同的计数——有状态的actor并没有使用可变量。由于这个actor是串行处理消息的，因此actor可以安全地访问其状态，而不会引发并发问题。

用API隐藏消息细节

Counter已经可以使用了，但并不方便。我们需要记住传给spawn()的参数是什么以及消息的细节（是{:next}、:next还是{:increment}？）。为了避免这些麻烦，可以将spawn()的调用和消息的细节全部隐藏到一组API函数中：

Actors/counter/counter.ex

```
defmodule Counter do
➤ def start(count) do
➤   spawn(__MODULE__, :loop, [count])
➤ end
➤ def next(counter) do
➤   send(counter, {:next})
```

```
➤   end
    def loop(count) do
      receive do
        {:next} ->
          IO.puts("Current count: #{count}")
          loop(count + 1)
      end
    end
  end
```

start()的实现用到了伪变量_MODULE_，其值是当前模块的名字。这样的API让actor的使用变得更简洁并且不易出错：

```
iex(1)> counter = Counter.start(42)
#PID<0.44.0>
iex(2)> Counter.next(counter)
Current count: 42
{:next}
iex(3)> Counter.next(counter)
{:next}
Current count: 43
```

仅输出状态的actor没有什么实用性。下面来看看如何让两个actor进行双向通信，让一个actor可以获取另一个actor的状态。

双向通信

之前提到过，actor是异步发送消息的——发送者并不会被阻塞。那我们怎么获得一个消息的回复呢？比如在之前的Counter中，如果不输出actor的当前计数，而是将当前计数返回给发送者会怎样呢？

actor模型没有提供直接回复消息的机制，但我们可以自行解决：将发送进程的标识符包含在消息中。通过这个机制，消息的接收者可以回复消息：

Actors/counter/counter2.ex

```
  defmodule Counter do
    def start(count) do
      spawn(__MODULE__, :loop, [count])
    end
    def next(counter) do
➤     ref = make_ref()
➤     send(counter, {:next, self(), ref})
➤     receive do
➤       {:ok, ^ref, count} -> count
➤     end
    end
    def loop(count) do
      receive do
        {:next, sender, ref} ->
```

```
➤         send(sender, {:ok, ref, count})
          loop(count + 1)
      end
    end
  end
```

这个版本不再输出当前计数，而是将当前计数返回给发送者，返回的消息类似于下面的三元组：

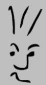
```
{:ok, ref, count}
```

ref是发送者用make_ref()生成的唯一引用。

来验证一下：

```
iex(1)> counter = Counter.start(42)
#PID<0.47.0>
iex(2)> Counter.next(counter)
42
iex(3)> Counter.next(counter)
43
```

现在可以为Counter的进程命名，这样就可以通过名称查找到对应的进程。



小乔爱问：

为什么回复的是一个元组?

在我们改造的 Counter 中可以不回复一个元组，而只回复计数即可：

```
{:next, sender} ->
  send(sender, count)
```

这是正确的，但 Elixir 习惯用元组作为消息，且第一个元素表示消息处理是成功的还是失败的。本例中的消息还带有发送者生成的唯一引用，这样如果多个消息到达信箱中，actor 就可以通过这个唯一引用来区分这些消息。

为进程命名

将一个消息发送给某个进程时，需要知道进程的标识符。如果是我们自己创建的进程，那不会有什么问题。但如果要向一个别人创建的进程发送消息呢?

这个问题可以用多种方法解决，最简单的方法就是为进程命名：

```
iex(1)> pid = Counter.start(42)
#PID<0.47.0>
iex(2)> Process.register(pid, :counter)
true
```

```
iex(3)> counter = Process.whereis(:counter)
#PID<0.47.0>
iex(4)> Counter.next(counter)
42
```

上面的代码用Process.register()为进程命名，并用Process.whereis()按名称查找进程。通过Process.registered()可以查看已被命名的所有进程：

```
iex(5)> Process.registered
[:kernel_sup, :init, :code_server, :user, :standard_error_sup,
 :global_name_server, :application_controller, :file_server_2, :user_drv,
 :kernel_safe_sup, :standard_error, :global_group, :error_logger,
 :elixir_counter, :counter, :elixir_code_server, :erl_prim_loader, :elixir_sup,
 :rex, :inet_db]
```

可以看到虚拟机在启动时已经命名了很多基本进程。之前使用send()函数时，其参数是进程变量，现在也可以使用进程名称：

```
iex(6)> send(:counter, {:next, self(), make_ref()})
{:next, #PID<0.45.0>, #Reference<0.0.0.107>}
iex(7)> receive do msg -> msg end
{:ok, #Reference<0.0.0.107>, 43}
```

继续简化Counter的API，使其不再使用进程变量作为参数：

Actors/counter/counter3.ex

```
def start(count) do
  pid = spawn(__MODULE__, :loop, [count])
➤ Process.register(pid, :counter)
  pid
end
def next do
  ref = make_ref()
➤ send(:counter, {:next, self(), ref})
  receive do
    {:ok, ^ref, count} -> count
  end
end
```

来验证一下：

```
iex(1)> Counter.start(42)
#PID<0.47.0>
iex(2)> Counter.next
42
iex(3)> Counter.next
43
```

今天的学习接近尾声。先进行一个茶歇，茶歇后我们将应用所学的知识来构造一个并行map函数，类似于Clojure的pmap。

茶歇——函数是第一类对象

与所有函数式语言一样，Elixir中的函数是第一类对象——函数可以被绑定到变量上，可以作为函数参数，与数据没什么区别。举例说明，用`iex`展示一下如何将匿名函数作为参数传给`Enum.map`，使数组中每个元素增倍：

```
iex(1)> Enum.map([1, 2, 3, 4], fn(x) -> x * 2 end)
[2, 4, 6, 8]
```

类似于Clojure的`#(...)`宏，Elixir也提供了定义匿名函数的快捷方式`&(...)`：

```
iex(2)> Enum.map([1, 2, 3, 4], &(&1 * 2))
[2, 4, 6, 8]
iex(3)> Enum.reduce([1, 2, 3, 4], 0, &(&1 + &2))
10
```

如果函数被绑定到了一个变量上，可以用`.`操作符来调用该变量代表的函数：

```
iex(4)> double = &(&1 * 2)
#Function<erl_eval.6.80484245>
iex(5)> double.(3)
6
```

再来看一个返回函数的函数：

```
iex(6)> twice = fn(fun) -> fn(x) -> fun.(fun.(x)) end end
#Function<erl_eval.6.80484245>
iex(7)> twice.(double).(3)
12
```

现在已经准备好了创建并行`map()`的所有工具，只剩下组装了。

并行map函数

之前已经用过了Elixir提供的`map()`函数，它可以对一个集合施加映射操作，不过是串行执行的。下面我们将其改造成能并行处理集合的每个元素：

Actors/parallel/parallel.ex

```
defmodule Parallel do
  def map(collection, fun) do
    parent = self()

    processes = Enum.map(collection, fn(e) ->
        spawn_link(fn() ->
            send(parent, {self(), fun.(e)})
          end)
      end)

    Enum.map(processes, fn(pid) ->
        receive do
```

```
          {^pid, result} -> result
        end
      end)
  end
end
```

这段代码分为两个阶段。第一阶段，为集合的每个元素创建一个进程（如果集合有1000个元素，将创建1000个进程）。每个进程对相应元素施加fun函数，并向发送消息的父进程回复施加函数的结果。第二阶段，父进程等待所有子进程的结果。

来验证一下：

```
iex(1)> slow_double = fn(x) -> :timer.sleep(1000); x * 2 end
#Function<6.80484245 in :erl_eval.expr/5>
iex(2)> :timer.tc(fn() -> Enum.map([1, 2, 3, 4], slow_double) end)
{4003414, [2, 4, 6, 8]}
iex(3)> :timer.tc(fn() -> Parallel.map([1, 2, 3, 4], slow_double) end)
{1001131, [2, 4, 6, 8]}
```

这段代码用到了:timer.tc()函数，其对参数函数的运行时间进行统计，并返回一个二元组，第一个元素是运行时间，第二个元素是参数函数的返回值。可以看到串行版本运行了4秒，而并行版本运行了1秒。

第一天总结

第一天的学习结束了。第二天我们将学习actor模型的错误处理和容错性。

第一天我们学到了什么

多个actor（进程）可以同时运行、不共享状态、通过向信箱异步地发送消息来进行通信。本章中我们学习如何实现下列任务：

- 用spawn()创建新进程；
- 用send()向进程发送消息；
- 通过模式匹配来处理消息；
- 连接两个进程，当一个进程结束时，另一个进程将接收到通知；
- 在异步通信的基础上，实现双向的同步通信；
- 为进程命名。

第一天自习

查找

- 阅读Elixir的函数库文档。
- 观看Erik Meijer、Clemens Szyperski和Carl Hewitt在Lang.NEXT 2012上关于actor模型的对话视频。

实践

- 测量在Erlang虚拟机上创建一个进程的成本，且与在JVM上创建一个线程的成本进行比较。
- 测量之前的并行map函数的成本，且与串行map函数的成本进行比较。何时应使用并行map函数，何时应使用串行map函数？
- 参考之前的并行map函数，写一个并行reduce函数。

5.3 第二天：错误处理和容错性

在1.3节已提到过：并发很重要的一个特性是并发代码具有容错性。今天就来学习actor模型提供的容错性。

不过首先要利用昨天的知识创建一个较复杂的贴近现实的例子，之后的讨论将在此基础上进行。

一个缓存actor

本节将创建一个网页缓存：向缓存添加页面时，需提供URL以及页面文本；向缓存请求页面时，需提供URL；也可以查看缓存一共包含了多少个字节。

我们需要一个字典，这个字典包含了URL到页面的映射。与Clojure的map类似，Elixir的字典是一个关联型的持久数据结构：

```
iex(1)> d = HashDict.new
#HashDict<[]>
iex(2)> d1 = Dict.put(d, :a, "A value for a")
#HashDict<[a: "A value for a"]>
iex(3)> d2 = Dict.put(d1, :b, "A value for b")
#HashDict<[a: "A value for a", b: "A value for b"]>
iex(4)> d2[:a]
"A value for a"
```

使用`HashDict.new`可以创建新的字典，使用`Dict.put(dict, key, value)`可以向其中添加元素，使用`dict[key]`可以从其中查找元素。

利用字典可以实现缓存：

Actors/cache/cache.ex

```
Line 1 defmodule Cache do
     -   def loop(pages, size) do
     -     receive do
     -       {:put, url, page} ->
     5         new_pages = Dict.put(pages, url, page)
     -         new_size = size + byte_size(page)
     -         loop(new_pages, new_size)
```

5

```
-       {:get, sender, ref, url} ->
-         send(sender, {:ok, ref, pages[url]})
10        loop(pages, size)
-       {:size, sender, ref} ->
-         send(sender, {:ok, ref, size})
-         loop(pages, size)
-       {:terminate} -> # 终止信号 - 终止递归
15    end
-   end
- end
```

这个缓存维护了两个状态：pages和size。pages是将URL映射到页面的字典；size是当前缓存的所有字节数（在第6行由byte_size()更新）。

与之前一样，我们仍用API来隐藏创建进程和发送消息的细节。下面的代码用于创建start_link()函数：

Actors/cache/cache.ex

```
def start_link do
  pid = spawn_link(__MODULE__, :loop, [HashDict.new, 0])
  Process.register(pid, :cache)
  pid
end
```

这段代码用空字典和0作为loop()的初始值，并将进程命名为:cache。下面的代码用于创建put()、get()、size()和terminate()函数：

Actors/cache/cache.ex

```
def put(url, page) do
  send(:cache, {:put, url, page})
end

def get(url) do
  ref = make_ref()
  send(:cache, {:get, self(), ref, url})
  receive do
    {:ok, ^ref, page} -> page
  end
end
def size do
  ref = make_ref()
  send(:cache, {:size, self(), ref})
  receive do
    {:ok, ^ref, s} -> s
  end
end

def terminate do
  send(:cache, {:terminate})
end
```

put()和terminate()函数只是简单地将参数包装在元组中，并将元组作为消息发送。而get()和size()函数比较复杂，因为它们需要等待一个消息的回复。本例中，这两个函数都在消息中带有唯一引用，正如我们昨天学过的那样。

来运行一下这个actor：

```
iex(1)> Cache.start_link
#PID<0.47.0>
iex(2)> Cache.put("google.com", "Welcome to Google ...")
{:put, "google.com", "Welcome to Google ..."}
iex(3)> Cache.get("google.com")
"Welcome to Google ..."
iex(4)> Cache.size()
21
```

一切顺利——现在已经可以向缓存中添加数据、取出数据，还能查看缓存的大小。

如果使用一些非法参数呢？比如使用nil作为页面：

```
iex(5)> Cache.put("paulbutcher.com", nil)
{:put, "paulbutcher.com", nil}
iex(6)>
=ERROR REPORT==== 22-Aug-2013::16:18:41 ===
Error in process <0.47.0> with exit value: {badarg,[{erlang,byte_size,[nil],[]} …

** (EXIT from #PID<0.47.0>) {:badarg, [{:erlang, :byte_size, [nil], []}, …
```


不出所料，因为没有检查参数，所以这次运行失败了。在大多数语言中，唯一的处理方法是添加一些检查参数的代码，当检查到非法参数时报错。Elixir提供了另一种方法——将错误处理隔离到一个管理进程中。这个方法看似简单，却是一个很大的改进，使代码更简洁、更具维护性，也更可靠。

在学习如何写管理进程之前，必须详细了解进程之间的连接。

错误检测

在5.2节中，我们用spawn_link()建立两个进程之间的连接，这样就可以检测到某一个进程的终止。连接是Elixir编程中最重要的概念之一——现在就来深入了解。

进程的异常终止通过连接进行传播

任何时候都可以用Process.link()在两个进程之间建立连接。下面定义一个简单的actor，用来讨论连接的原理：

Actors/links/links.ex

```
defmodule LinkTest do
  def loop do
    receive do
```

```
      {:exit_because, reason} -> exit(reason)
      {:link_to, pid} -> Process.link(pid)
      {:EXIT, pid, reason} -> IO.puts("#{inspect(pid)} exited because #{reason}")
    end
    loop
  end
end
```

现在创建这个actor的两个实例，将这两个进程连接起来，并让其中一个异常终止：

```
iex(1)> pid1 = spawn(&LinkTest.loop/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
iex(3)> send(pid1, {:link_to, pid2})
{:link_to, #PID<0.49.0>}
iex(4)> send(pid2, {:exit_because, :bad_thing_happened})
{:exit_because, :bad_thing_happened}
```

这段代码首先创建了actor的两个实例，将其进程标识分别绑定到`pid1`和`pid2`。然后创建从`pid1`到`pid2`的连接。最后让`pid2`的进程异常终止。

`pid1`本应打印`pid2`异常终止的原因，但我们注意到`pid1`并没有输出，原因是没有设置`:trap_exit`。另外，如果用`Process.info()`查看两个进程的状态，会看到以下现象：

```
iex(5)> Process.info(pid2, :status)
nil
iex(6)> Process.info(pid1, :status)
nil
```

这样一来不只是`pid2`终止，而是两个进程都终止了。我们先来做另一个试验，再来学习如何修复这个问题。

连接是双向的

如果重复上面的试验，让`pid1`终止，就会看到同样的结果——两个进程都终止了：

```
iex(1)> pid1 = spawn(&LinkTest.loop/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
iex(3)> send(pid1, {:link_to, pid2})
{:link_to, #PID<0.49.0>}
iex(4)> send(pid1, {:exit_because, :another_bad_thing_happened})
{:exit_because, :another_bad_thing_happened}
iex(5)> Process.info(pid1, :status)
nil
iex(6)> Process.info(pid2, :status)
nil
```

可见连接是双向的。建立了从`pid1`到`pid2`的连接的同时，也就建立了从`pid2`到`pid1`的连接——所以如果其中一个进程终止，那么两个进程就都终止了。

正常终止

如果尝试让已经连接的一个进程正常终止（用:normal这个理由退出进程），会观察到以下现象：

```
iex(1)> pid1 = spawn(&LinkTest.loop/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
iex(3)> send(pid1, {:link_to, pid2})
{:link_to, #PID<0.49.0>}
iex(4)> send(pid2, {:exit_because, :normal})
{:exit_because, :normal}
iex(5)> Process.info(pid2, :status)
nil
iex(6)> Process.info(pid1, :status)
{:status, :waiting}
```

可见进程正常终止是不会让连接的另一个进程终止的。

系统进程

通过设置进程的:trap_exit标识，可以让一个进程捕获另一个进程的终止消息。用专业术语来说，这是将进程转化为*系统进程*：

Actors/links/links.ex

```
def loop_system do
  Process.flag(:trap_exit, true)
  loop
end
```

来测试一下：

```
iex(1)> pid1 = spawn(&LinkTest.loop_system/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
iex(3)> send(pid1, {:link_to, pid2})
{:link_to, #PID<0.49.0>}
iex(4)> send(pid2, {:exit_because, :yet_another_bad_thing_happened})
{:exit_because, :yet_another_bad_thing_happened}
#PID<0.49.0> exited because yet_another_bad_thing_happened
iex(5)> Process.info(pid2, :status)
nil
iex(6)> Process.info(pid1, :status)
{:status, :waiting}
```

现在，可以用loop_system启动pid1。当pid2终止时，pid1会得到消息（并打印退出的消息），并且会继续运行。

管理进程

我们已经准备好实现一个进程管理者（也就是一个系统进程），它管理着若干个工作进程，当工作进程崩溃时进行干预。

下面的代码为前述的缓存actor创建一个管理者，当缓存进程崩溃时，管理者会将其重启：

Actors/cache/cache.ex

```
defmodule CacheSupervisor do
  def start do
    spawn(__MODULE__, :loop_system, [])
  end
  def loop do
    pid = Cache.start_link
    receive do
      {:EXIT, ^pid, :normal} ->
        IO.puts("Cache exited normally")
        :ok
      {:EXIT, ^pid, reason} ->
        IO.puts("Cache failed with reason #{inspect reason} - restarting it")
        loop
    end
  end
  def loop_system do
    Process.flag(:trap_exit, true)
    loop
  end
end
```

这个actor将自己转化为系统进程，并进入`loop()`。`loop()`创建了`Cache.loop()`进程，并一直阻塞，直到所创建的进程终止。该进程若是正常终止，则管理者也正常终止（返回`:ok`），否则`loop()`进行递归并重新创建缓存进程。

现在不必直接启动`Cache`实例，而是启动`CacheSupervisor`，其将负责创建`Cache`实例：

```
iex(1)> CacheSupervisor.start
#PID<0.47.0>
iex(2)> Cache.put("google.com", "Welcome to Google ...")
{:put, "google.com", "Welcome to Google ..."}
iex(3)> Cache.size
21
```

如果缓存崩溃，就会被自动重启：

```
iex(4)> Cache.put("paulbutcher.com", nil)
{:put, "paulbutcher.com", nil}
Cache failed with reason {:badarg, [{:erlang, :byte_size, [nil], []}, …
iex(5)>
=ERROR REPORT==== 22-Aug-2013::17:49:24 ===
Error in process <0.48.0> with exit value: {badarg,[{erlang,byte_size,[nil],[]}, …

iex(5)> Cache.size
0
```

```
iex(6)> Cache.put("google.com", "Welcome to Google ...")
{:put, "google.com", "Welcome to Google ..."}
iex(7)> Cache.get("google.com")
"Welcome to Google ..."
```

虽然缓存崩溃时我们会丢失之前的数据，但至少得到了一个崩溃后可以继续使用的缓存。

超时

将缓存自动重启是个不错的方法，但并不是万能药。如果两个进程同时向缓存发送消息，下面这些事件会依次发生：

(1) 进程1向缓存发送`:put`消息；
(2) 进程2向缓存发送`:get`消息；
(3) 缓存在处理进程1的消息时崩溃了；
(4) 管理者将缓存重启，但进程2的消息丢失了；
(5) 进程2在`receive`处陷入死锁，一直在等待消息的回复，但这个回复永远不会发送。

我们可以用`after`语句来为`receive`增加超时机制，这需要修改一下`get()`（`size()`函数也需要同样的修改）：

Actors/cache/cache2.ex

```
def get(url) do
  ref = make_ref()
  send(:cache, {:get, self(), ref, url})
  receive do
    {:ok, ^ref, page} -> page
➤   after 1000 -> nil
  end
end
```



小乔爱问：

消息是否保证能被送达?

造成上面现象的主要原因是缓存重启时消息丢失了，这就牵扯到一个很重要的问题——Elixir是否能确保消息一定能被送达并被处理?

Elixir有两个规则：

- 如果没有异常发生，消息一定能被送达并被处理；
- 如果某个环节出现异常，异常一定会通知到使用者（假设使用者已经连接到或正在管理发生异常的进程）

第二条规则是Elixir提供容错性的基石。

错误处理内核（error-Kernel）模式

Tony Hoare有一句名言[1]。

> 软件设计有两种方式：一种方式是，使软件过于简单，明显地没有缺陷；另一种方式是，使软件过于复杂，没有明显的缺陷。

actor提供了一种容错的方式：错误处理内核模式，在两者之间找到了平衡。

一个软件系统如果应用了错误处理内核模式，那么该系统正确运行的前提是其错误处理内核必须正确运行。成熟的程序通常使用尽可能小而简单的错误处理内核——小而简单到明显地没有缺陷。

对于一个使用actor模型的程序，其错误处理内核是顶层的管理者，管理着子进程——对子进程进行启动、停止、重启等操作。

程序的每个模块都有自己的错误处理内核——模块正确运行的前提是其错误处理内核必须正确运行。子模块也会有自己的错误处理内核，以此类推。这就构成了错误处理内核的层级树，较危险的操作都会被下放给底层的actor执行，如图5-2所示。

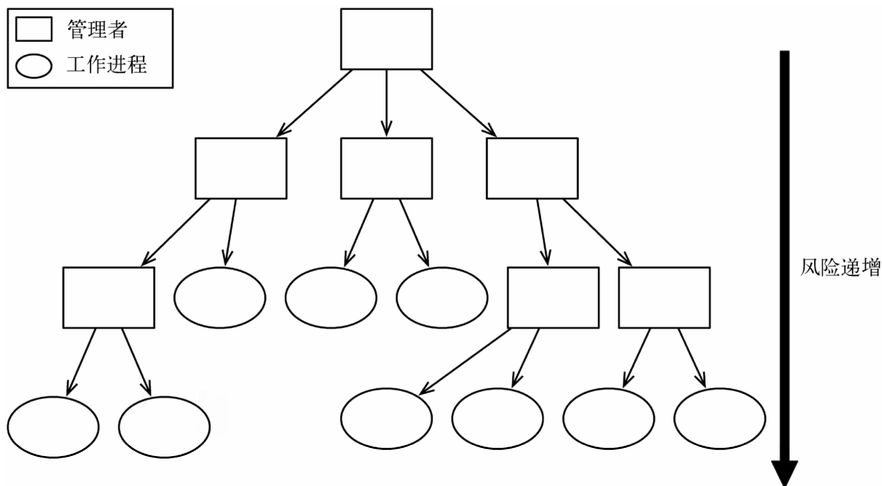


图5-2　错误处理的层级树

错误处理内核模式主要解决了防御式编程中碰到的一些棘手问题。

任其崩溃

防御式编程主要通过预言可能出现的缺陷来实现容错性。举例说明，假设有一个函数，其接

① http://zoo.cs.yale.edu/classes/cs422/2011/bib/hoare81emperor.pdf

受一个字符串，当字符串全是大写时返回true，否则返回false。先草拟一个版本：

```
def all_upper?(s) do
  String.upcase(s) == s
end
```

看上去不错，但如果传入nil作为参数，这段代码将崩溃。为了解决这个问题，有些程序员就对其进行了如下修改：

```
defmodule Upper do
  def all_upper?(s) do
    cond do
      nil?(s) -> false
      true -> String.upcase(s) == s
    end
  end
end
```

现在再碰到nil作为参数时，代码就不会崩溃了，但如果传入另一些不按套路出牌的参数呢（比如一个关键字）？使用nil作为参数对这个函数是否真的有意义？这样修改代码很有可能会引发一个缺陷——只是我们暂时隐藏了这个缺陷，之后也很难意识到其存在，但总有一天它会跳起来咬我们一口。

使用actor模型的程序并不进行防御式编程，而是遵循"任其崩溃"的哲学，让actor的管理者来处理这些问题。这样做有几个好处，比如：

- 代码会变得更加简洁且容易理解，可以清晰区分出"一帆风顺"的代码和容错代码；
- 多个actor之间是相互独立的，并不共享状态，因此一个actor的崩溃不太会殃及到其他actor。尤其重要的是一个actor的崩溃不会影响到其管理者，这样管理者才能正确处理此次崩溃；
- 管理者也可以选择不处理崩溃，而是记录崩溃的原因，这样我们就会得到崩溃通知并进行后续处理。

虽然第一眼看上去"任其崩溃"的哲学有点奇怪，但它和错误处理内核模式都在产品环境上反复进行过验证。一些系统的可用性据说提高到了99.9999999%（9个9，参见*Programming Erlang: Software for a Concurrent World* [Arm13][①]）

第二天总结

我们第一天学习了actor模型的基础知识，第二天学习了actor模型如何进行容错。第三天将学习如何用actor模型进行分布式编程。

① 该书中文版《Erlang程序设计（第2版）》已由人民邮电出版社出版：http://www.ituring.com.cn/book/1264。
——编者注

第二天我们学到了什么

Elixir通过创建管理者并使用进程的连接来进行容错：

- 连接是双向的——如果进程a连接到进程b，那么进程b也连接到进程a；
- 连接可以传递错误——如果两个进程已经连接，其中一个进程异常终止，那么另一个进程也会异常终止；
- 如果进程被转化成系统进程，当其连接的进程异常终止时，系统进程不会终止，而是会收到`:EXIT`消息。

第二天自习

查找

- 阅读`Process.monitor()`的相关文档——管理一个进程与连接一个进程有什么区别？何时使用管理，何时使用连接？
- Elixir是如何进行异常处理的？何时应该使用异常处理，而不使用管理者和“任其崩溃”的哲学呢？

实践

- 在`receive`块中，如果一个消息没法匹配到任何模式，将会被留在进程的信箱中。利用这个特性和超时特性，实现一个*有优先级的信箱*，即使低优先级的消息可能比高优先级的消息更早地被接收，但高优先级的消息会比低优先级的消息更早地被处理。
- 改进今天开篇介绍的缓存actor，根据hash函数将缓存元素分配到多个actor中。创建一个管理actor，其负责创建多个工作actor，并将消息转发给对应的工作actor。如果一个工作actor崩溃，管理actor应该怎么办？

5.4　第三天：分布式

到目前为止我们学习的所有知识都只支持一台计算机，相比于已经学习过的并发模型，actor模型的一个很大的优点是其支持分布式——它可以将消息发送到另一台计算机上的actor，就像发送到本地计算机上的actor一样。

讨论分布式之前，要了解Elixir提供的一个强大的工具——OTP。

OTP

过去两天，我们都在用“原始”的Elixir进行演示。这有利于我们理解其运行机制，但如果用原始的方式创建每一个工作进程和管理者，那就会变得非常无趣且容易出错。讲到这里你肯定猜到了我们将要介绍一个能解决这个问题的库——OTP。



小乔爱问：

OTP代表了什么？

缩写单词通常只为自己代言。IBM 字面上是 International Business Machines 的缩写，但对于大多数人来说 IBM 就是 IBM：缩写就是约定俗成的名称。类似地，BBC 也不再是 British Broadcasting Corporation 的缩写，OTP 也不再是 Open Telecom Platform 的缩写。

Erlang（包括 Elixir）最初是从电信业发展起来的，许多 Erlang 的最佳实践都被收录到了 OTP 中。但 OTP 不是电信业专用的，OTP 就是 OTP。

在学习OTP之前，先来看看在Elixir中函数与模式匹配是如何交互的。

函数与模式匹配

之前我们仅在介绍`receive`块时提到过模式匹配，然而在Elixir中模式匹配随处可见。值得强调的是，每次调用函数时，其实都在进行模式匹配。用一个简单的函数来举例：

Actors/patterns/patterns.ex

```
defmodule Patterns do
  def foo({x, y}) do
    IO.puts("Got a pair, first element #{x}, second #{y}")
  end
end
```

这段代码定义了一个函数，其接受一个参数，并用模式`{x, y}`匹配这个参数。如果用一个二元组进行匹配，那么二元组的第一个元素会绑定到x上，第二个元素会绑定到y上：

```
iex(1)> Patterns.foo({:a, 42})
Got a pair, first element a, second 42
:ok
```

如果用一个不匹配的参数进行调用，将会得到一个错误：

```
iex(2)> Patterns.foo("something else")
** (FunctionClauseError) no function clause matching in Patterns.foo/1
    patterns.ex:3: Patterns.foo("something else")
    erl_eval.erl:569: :erl_eval.do_apply/6
    src/elixir.erl:147: :elixir.eval_forms/3
```

根据需要可以为一个函数添加多个不同的定义：

Actors/patterns/patterns.ex

```
def foo({x, y, z}) do
  IO.puts("Got a triple: #{x}, #{y}, #{z}")
end
```

5

调用函数时，与参数匹配的函数将被执行：

```
iex(2)> Patterns.foo({:a, 42, "yahoo"})
Got a triple: a, 42, yahoo
:ok
iex(3)> Patterns.foo({:x, :y})
Got a pair, first element x, second y
:ok
```

下面将使用OTP实现一个服务器，其中也用到了这个技术。

用GenServer重新实现缓存

现在学习OTP的一个组件：GenServer。GenServer是一个行为（behaviour），可以用来自动创建一个有状态的actor。我们就来利用这个组件重新实现昨天的缓存。

你可能觉得behaviour这种拼写有点怪，它来源于Erlang，而Erlang用的是英式拼写。我们就入乡随俗沿用这种拼写。

这里所说的“行为”非常类似于Java中的接口——其定义了一个函数集。模块使用`use`来声明自己实现了行为：

Actors/cache/cache3.ex

```
defmodule Cache do
➤ use GenServer.Behaviour
  def handle_cast({:put, url, page}, {pages, size}) do
    new_pages = Dict.put(pages, url, page)
    new_size = size + byte_size(page)
    {:noreply, {new_pages, new_size}}
  end
  def handle_call({:get, url}, _from, {pages, size}) do
    {:reply, pages[url], {pages, size}}
  end

  def handle_call({:size}, _from, {pages, size}) do
    {:reply, size, {pages, size}}
  end
end
```

这段代码中`Cache`声明自己实现了一个行为（`GenServer.Behaviour`）和两个函数（`handle_cast()`和`handle_call()`）。

`handle_cast()`可以处理消息但并不回复消息。其接受两个参数：收到的消息、actor的当前状态。返回值是一个二元组`{:noreply, new_state}`。本例中实现了一个`handle_cast()`来处理`:put`消息。

`handle_call()`可以处理消息且回复消息。其接受三个参数：收到的消息、发送者标识、actor的当前状态。返回值是一个三元组`{:reply, reply_value, new_state}`。本例中实现了两个`handle_call()`，一个负责处理`:get`消息，另一个负责处理`:size`消息。类似于Clojure，Elixir

用下划线（_）开头的变量名来表示该变量不被使用——比如_from。

按照惯例仍会提供一些便于使用的API：

Actors/cache/cache3.ex

```
def start_link do
  :gen_server.start_link({:local, :cache}, __MODULE__, {HashDict.new, 0}, [])
end

def put(url, page) do
  :gen_server.cast(:cache, {:put, url, page})
end

def get(url) do
  :gen_server.call(:cache, {:get, url})
end

def size do
  :gen_server.call(:cache, {:size})
end
```

这段代码使用了`:gen_server.start_link()`替换`spawn_link()`，使用`:gen_server.cast()`发送一个不需要回复的消息，使用`:gen_server.call()`发送一个需要回复的消息。

接下来，用OTP创建一个管理者。

OTP管理者

这是一个用OTP管理者行为来实现的缓存管理者：

Actors/cache/cache3.ex

```
defmodule CacheSupervisor do
  def init(_args) do
    workers = [worker(Cache, [])]
    supervise(workers, strategy: :one_for_one)
  end
end
```

进程启动时会调用`init()`函数。其接受一个参数（本例中没有使用这个参数），创建一些工作进程并将其管理起来。本例中创建了一个`Cache`进程，并使用`one-for-one`重启策略来管理该进程。



小乔爱问：

什么是重启策略？

OTP 管理者行为支持多种不同的重启策略，最常用的是 one-for-all 和 one-for-one。

这些策略指的是管理多个工作进程的管理者如何重启崩溃的工作进程。如果一个工作进

程崩溃，使用 one-for-all 策略的管理者将重启所有工作进程（包括那些没有崩溃的工作进程）。使用 one-for-one 策略的管理者仅重启已经崩溃的工作进程。

还有许多其他的策略，不过这两种已经可以应对大部分场景了。

按照惯例，提供易用的API：

Actors/cache/cache3.ex

```
def start_link do
  :supervisor.start_link(__MODULE__, [])
end
```

你可以自行验证一下缓存和管理者是否能正常工作，在昨天的学习中进行过类似的验证，此处不再赘述。



小乔爱问：

OTP还能做什么？

正如以上的代码所示，OTP 可以帮我们省去一些无聊的代码。此外它还提供了更多的好处，这些好处在之前的例子中不是那么明显。比起之前创建的简单版本，用 OTP 实现的服务器和管理者有着更多的功能，其中包括以下几点。

更好的重启逻辑：之前我们自己实现的简单管理者使用非常草率的重启策略——如果工作线程崩溃，就将其重启。如果工作线程在启动时很快就崩溃，那么管理者会一直重启工作线程。而 OTP 提供的管理者可以设定最大重启频率，如果重启超过这个频率，管理者将会异常终止。

调试与日志：通过调整 OTP 服务器的参数，可以开启调试和日志功能，这对开发很重要。

代码热升级：OTP 服务器不需要停止整个系统就可以进行升级。

还有许多：发布管理、故障切换、自动扩容，等等。

本书不会详细介绍这些特性。在大部分场景中可以直接使用这些特性，而不建议自己造轮子。

节点

每创建一个Erlang虚拟机实例，就相当于创建了一个节点。之前的例子都只创建了一个节点。现在来学习如何创建和连接多个节点。

连接（connect）[①]节点

连接两个节点时，必须先对这两个节点命名。启动Erlang虚拟机时可以使用--name或者--sname选项为节点命名。我的MacBook Pro的IP是10.99.1.50。运行iex --sname node1@10.99.1.50 --cookie yumyum（稍后会解释--cookie参数），可以查看节点名称：

```
iex(node1@10.99.1.50)1> Node.self
:"node1@10.99.1.50"
iex(node1@10.99.1.50)2> Node.list
[]
```

使用Node.self()可以查看节点名称，使用Node.list()可以查看当前节点已知的其他节点列表。现在这个列表是空的，下面将为其赋值。如果使用iex --sname node2@10.99.1.92 --cookie yumyum在另一台计算机（10.99.1.92）上运行另一个Erlang虚拟机，用Node.connect()可以连接该节点：

```
iex(node1@10.99.1.50)3> Node.connect(:"node2@10.99.1.92")
true
iex(node1@10.99.1.50)4> Node.list
[:"node2@10.99.1.92"]
```

连接是双向的，第二台计算机也知道第一台的信息：

```
iex(node2@10.99.1.92)1> Node.list
[:"node1@10.99.1.50"]
```




小乔爱问：

如果只有一台计算机呢？

如果你只有一台计算机，但又想进行集群试验，那有以下几种选择：

- 使用虚拟机；
- 使用Amazon EC2或者类似的云服务；
- 在一台计算机上运行多个节点。虽然这种方案与实际环境有一些偏差，却是目前最简单的方案。如果你对如何配置多机环境不太熟悉，这种方式可以帮你避免设置防火墙和配置网络等麻烦。

远程执行

已经建立了两个连接的节点，一个节点可以在另一个节点上执行代码：

```
iex(node1@10.99.1.50)5> whoami = fn() -> IO.puts(Node.self) end
#Function<20.80484245 in :erl_eval.expr/5>
```

① 之前我们看到过对进程的连接，其指的是link；此处的连接指的是connect，这两个概念不可混淆。——译者注

```
iex(node1@10.99.1.50)6> Node.spawn(:"node2@10.99.1.92", whoami)
#PID<8242.50.0>
node2@10.99.1.92
```

这段看似简单的代码却异常强大——一个节点在另一个节点上执行代码，而且执行的结果还会返回给第一个节点。这是因为子进程会继承父进程的组长（group leader），IO.puts()会将输出发送给组长。其暗地里进行了很多处理！

远程消息

如你所料，一个actor可以向另一台计算机的actor发送消息。举例说明，下面的代码在一个节点上创建了一个Counter的实例（参见5.2节的“有状态的actor”部分）：

```
iex(node2@10.99.1.92)1> pid = spawn(Counter, :loop, [42])
#PID<0.51.0>
iex(node2@10.99.1.92)2> :global.register_name(:counter, pid)
:yes
```

创建好实例后，用:global.register_name()进行注册，这类似于Process.register()，但它是在集群全局注册名字。

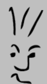
现在就可以在另一个节点上使用:global.whereis_name()获取进程标识符，并发送消息：

```
iex(node1@10.99.1.50)1> Node.connect(:"node2@10.99.1.92")
true
iex(node1@10.99.1.50)2> pid = :global.whereis_name(:counter)
#PID<7856.51.0>
iex(node1@10.99.1.50)3> send(pid, {:next})
{:next}
iex(node1@10.99.1.50)4> send(pid, {:next})
{:next}
```

显然，在第一个节点上的输出是：

```
Current count: 42
Current count: 43
```

重申一下：消息的运行结果会输出到产生消息的actor的父进程节点上。



小乔爱问：

我该如何管理集群？

一个节点可以在另一个节点上远程执行代码，这是非常强大的一个功能。不过强大的功能都很危险。在设计集群管理策略时尤其需要考虑安全性。之前调用 iex 时使用的--cookie 参数就源出于此——一个 Erlang 节点仅接收使用同样 cookie 的节点发送的消息。也有其他的方法用来保障 Erlang 集群的安全性，比如 SSL 隧道连接。

安全性不是唯一的问题。上面的例子中使用 IP 地址作为节点名称的一部分，这在大部

分场景下都适用（因为我并不知道你的网络配置，使用IP比较保险）。不过在产品环境中未必是最好的选择。

集群设计中的种种权衡非常复杂，也超出了本书的范围。在产品环境中使用集群前，请务必阅读相关文档。

分布式词频统计

我们即将结束对actor模型和Elixir的学习，现在来尝试实现分布式的Wikipedia词频统计（前几章已经介绍过其背景）。分布式的解决方案与前几章的解决方案相比，相同的是可以借助多核的力量；不同的是它还可以利用多台计算机的力量，且能从崩溃中恢复。

分布式解决方案的基本架构如图5-3所示。

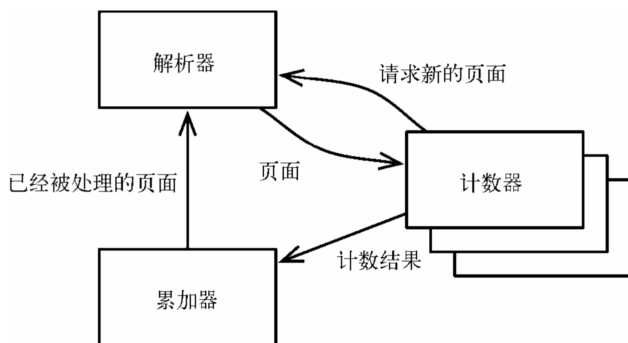


图5-3 分布式解决方案的基本架构

分布式解决方案涉及到三类actor：一个解析器（`Parser`），多个计数器（`Counter`）和一个累加器（`Accumulator`）。解析器负责将一个Wikipedia dump解析成若干个页面，计数器负责统计页面的词频，累加器负责统计多个页面的词频总数。

处理的第一步是计数器向解析器请求一个页面。计数器收到页面后，统计页面的词频，并将结果传给累加器。累加器处理完成后，会告诉解析器该页面已经被处理。

我们稍后会解释为什么要选择这样的处理流程，先来看看如何实现这个方案，从计数器的部分开始。

计数器

下面的`Counter`模块是一个简单的无状态的actor，从`Parser`接收页面，并将计数结果发送给`Accumulator`：

Actors/word_count/lib/counter.ex

```
Line 1 defmodule Counter do
     -   use GenServer.Behaviour
     -   def start_link do
     -     :gen_server.start_link(__MODULE__, nil, [])
     5   end
     -   def deliver_page(pid, ref, page) do
     -     :gen_server.cast(pid, {:deliver_page, ref, page})
     -   end
     -
    10   def init(_args) do
     -     Parser.request_page(self())
     -     {:ok, nil}
     -   end
     -
    15   def handle_cast({:deliver_page, ref, page}, state) do
     -     Parser.request_page(self())
     -
     -     words = String.split(page)
     -     counts = Enum.reduce(words, HashDict.new, fn(word, counts) ->
    20         Dict.update(counts, word, 1, &(&1 + 1))
     -       end)
     -     Accumulator.deliver_counts(ref, counts)
     -     {:noreply, state}
     -   end
    25 end
```

这段代码遵循了OTP服务器的标准模式——首先是供外部使用的API（`start_link()`和`deliver_page()`），然后是初始化函数（`init()`），最后是消息处理函数（`handle_cast()`）。

`Counter`初始化时先调用`Parser.request_page()`（第11行）。

`Counter`每接收到一个页面，首先会申请下一个页面（第16行，这样做是为了减少延迟）。然后统计当前页面的词频，构造字典`counts`来保存结果（第18~21行）。最后将`ref`（引用`ref`是和页面一起接收到的）和计数结果发送给`Accumulator`。

利用`CounterSupervisor`可以创建和管理多个`Counter`：

Actors/word_count/lib/counter.ex

```
defmodule CounterSupervisor do
  use Supervisor.Behaviour
  def start_link(num_counters) do
    :supervisor.start_link(__MODULE__, num_counters)
  end
  def init(num_counters) do
    workers = Enum.map(1..num_counters, fn(n) ->
      worker(Counter, [], id: "counter#{n}")
    end)
    supervise(workers, strategy: :one_for_one)
  end
end
```

CounterSupervisor.init()的参数是要创建的Counter的个数，也是workers列表的长度。注意，每个工作线程worker都需要一个唯一的id，可以用1..num_counters构造id的值。

累加器

Accumulator维护了两个状态：totals是保存累加结果的字典；processed_pages是保存已经处理过的页面所对应的引用的集合。

Actors/word_count/lib/accumulator.ex

```
Line 1 defmodule Accumulator do
     -   use GenServer.Behaviour
     -
     -   def start_link do
     5     :gen_server.start_link({:global, :wc_accumulator}, __MODULE__,
     -       {HashDict.new, HashSet.new}, [])
     -   end
     -
     -   def deliver_counts(ref, counts) do
    10     :gen_server.cast({:global, :wc_accumulator}, {:deliver_counts, ref, counts})
     -   end
     -
     -   def handle_cast({:deliver_counts, ref, counts}, {totals, processed_pages}) do
     -     if Set.member?(processed_pages, ref) do
    15       {:noreply, {totals, processed_pages}}
     -     else
     -       new_totals = Dict.merge(totals, counts, fn(_k, v1, v2) -> v1 + v2 end)
     -       new_processed_pages = Set.put(processed_pages, ref)
     -       Parser.processed(ref)
    20       {:noreply, {new_totals, new_processed_pages}}
     -     end
     -   end
     - end
```

这段代码以{:global, wc_accumulator}为参数调用:gen_server.start_link()（第5行），为累加器创建了全局名称。可以直接使用全局名称来调用:gen_server.cast()发送消息（第10行）。

当Accumulator收到计数结果时，首先检查某一页面是否已经被处理过（稍后我们会看到Accumulator可能收到两次同一页面的计数结果，并解释这个检查的重要性）。如果页面没有被处理过，则使用Dict.merge()将该页面的计数结果合并到totals中，并使用Set.put()将该页面的引用合并到processed_pages，最后通知Parser该页面已经被处理。

解析器与容错

解析器是三种actor中最复杂的，我们将逐步进行介绍。首先，介绍其对外提供的API：

Actors/word_count/lib/parser.ex

```
defmodule Parser do
  use GenServer.Behaviour
```

5

```
  def start_link(filename) do
    :gen_server.start_link({:global, :wc_parser}, __MODULE__, filename, [])
  end

  def request_page(pid) do
    :gen_server.cast({:global, :wc_parser}, {:request_page, pid})
  end

  def processed(ref) do
    :gen_server.cast({:global, :wc_parser}, {:processed, ref})
  end
end
```

与Accumulator相同，Parser也在初始化时注册了一个全局名称。它提供了两种操作——第一种是request_page()，Counter调用这个函数来请求一个页面；第二种是processed()，Accumulator调用这个函数来通知解析器某页面已经被处理了。

接下来，介绍这两种操作的消息处理函数：

Actors/word_count/lib/parser.ex

```
def init(filename) do
  xml_parser = Pages.start_link(filename)
  {:ok, {ListDict.new, xml_parser}}
end

def handle_cast({:request_page, pid}, {pending, xml_parser}) do
  new_pending = deliver_page(pid, pending, Pages.next(xml_parser))
  {:noreply, {new_pending, xml_parser}}
end

def handle_cast({:processed, ref}, {pending, xml_parser}) do
  new_pending = Dict.delete(pending, ref)
  {:noreply, {new_pending, xml_parser}}
end
```

Parser维护了两个状态：第一个是pending，这是一个ListDict，其元素是已经发往Counter但没有被处理的页面的引用；第二个是xml_parser，这是一个actor，其使用Erlang的xmerl库[1]来解析Wikipedia dump（在此不详述其实现，详情可见本书配套代码）。

对:processed消息的处理只需要从pending中删除已被处理的页面。对:request_page消息的处理需要从XML解析器中获取下一个可用的页面，并将页面传给deliver_page()：

Actors/word_count/lib/parser.ex

```
defp deliver_page(pid, pending, page) when nil?(page) do
  if Enum.empty?(pending) do
    pending # 什么也不做
```

① http://www.erlang.org/doc/apps/xmerl/

```
    else
      {ref, prev_page} = List.last(pending)
      Counter.deliver_page(pid, ref, prev_page)
      Dict.put(Dict.delete(pending, ref), ref, prev_page)
    end
  end

  defp deliver_page(pid, pending, page) do
    ref = make_ref()
    Counter.deliver_page(pid, ref, page)
    Dict.put(pending, ref, page)
  end
```

这里的deliver_page()使用到了一个未介绍过的Elixir特性——**卫语句**（guard clause），在本例中是第一个deliver_page()的when子句。卫语句是一个布尔表达式——函数仅在表达式为真时有效。

当page不为空时，首先使用make_ref()创建一个唯一的引用，并将页面传给请求页面的counter，最后将页面添加到pending中。

当page为空时，意味着XML解析器已经完成了对所有页面的解析，不再提供新的页面。此时只需将pending中最老的元素传给Counter，并从pending中将此页面移出再重新添加进来，以保证其是pending中最新的元素。

page为空的分支主要是为了确保pending中的页面最终都会被处理。将这些pending中的页面再次发给另一个Counter有什么好处吗？

和牌了

这样的好处是提升了容错性。如果一个Counter崩溃、网络故障或者硬件故障，就需要将其处理的页面发给另一个Counter。由于每个页面都带有唯一的引用，那就可以分辨哪些页面已经被处理过了，从而避免重复计数。

现在启动一个集群来体验一下吧。在一台计算机上启动一个Parser和一个Accumulator，并在其他的一台或几台计算机上启动几个Counter。如果拔掉某个运行Counter的计算机的网线，或者干掉其Erlang虚拟机，其他正常的Counter将继续运行并接管那些运行在故障计算机上的页面。

这是一个体现并发分布式程序的优点的绝佳例子。发生某个硬件故障时，串行的程序或多线程的程序都会崩溃，但这个分布式的程序将幸存下来。

第三天总结

我们完成了第三天的学习，同时也结束了对actor模型的学习。

第三天我们学到了什么

使用Elixir可以创建多节点的集群。一个节点上的actor可以向另一个节点上的actor发送消息，

与向本节点的actor发送消息没有什么区别。使用Elixir可以创建一个分布在多台计算机上的系统，如果其中一台计算机崩溃，该系统可以从中恢复运行。

第三天自习

查找

- 观看Joe Armstrong在Lambda Jam上的演讲：Systems That Run Forever Self-Heal and Scale。
- 什么是一个OTP应用程序？为什么把它理解成“组件”更加合适？
- 到目前为止，我们使用的actor的状态都会在它退出时丢失。Elixir是如何实现持久状态的？

实践

- 对于本节中具有容错功能的词频统计程序，如果一个计数器或相应的计算机崩溃，程序可以继续运行下去。但如果一个解析器或一个累加器崩溃则不行。修改程序，使其在任一actor或任一计算机崩溃时都可以继续运行。

5.5　复习

Smalltalk的设计者、面向对象编程之父Alan Kay曾经这样描述面向对象的本质[①]：

> 很久以前，我在描述“面向对象编程”时使用了“对象”这个概念。很抱歉这个概念让许多人误入歧途，他们将学习的重心放在了“对象”这个次要的方面。
>
> 真正主要的方面是“消息”……日文中有一个词ma，表示“间隔”，与其最为相近的英文或许是“interstitial”。创建一个规模宏大且可生长的系统的关键在于其模块之间应该如何交流，而不在于其内部的属性和行为应该如何表现。

这段话也概括了使用actor模型进行编程的精髓——我们可以认为actor模型是面向对象模型在并发编程领域的扩展。actor模型精心设计了消息传输和封装的机制，强调了面向对象的精髓，可以说actor模型非常“面向对象”。

优点

actor有许多优良的特性，适用于解决多种并发问题。

消息传输和封装

虽然多个actor可以同时运行，但它们并不共享状态，而且在单个actor中所有事件都是串行执行的。所以关于并发，只需要关注于多个actor之间的消息流即可。

① http://c2.com/cgi/wiki?AlanKayOnMessaging

对开发人员来说这是个重大利好。每个actor可以被单独测试，而且当测试覆盖了某个actor的消息类型和消息顺序时，就可以确定这个actor非常可靠。如果发现了一个与并发相关的bug，也就知道重点应该放在actor之间的消息流上。

容错

使用actor模型的程序天生具有容错性。这不仅会让程序更加强壮，而且（通过“任其崩溃”的哲学）会让代码更加简洁明了。

分布式编程

actor模型支持共享内存模型，也支持分布式内存模型，这就带来了很多优点。

首先，actor模型几乎可以解决任何规模的问题。我们不需要将问题局限于用一个系统解决。

其次，actor模型可以解决地理分布式问题。对于不同部分需要部署在不同地理位置的软件，Actor模型是个极佳的选择。

最后，分布式是软件具有容错能力的基石。

缺点

尽管使用actor模型的程序比使用线程与锁模型的程序更容易debug，但actor模型仍会碰到死锁这一类的共性问题，也会碰到一些actor模型独有的问题（例如信箱溢出）。

类似于线程与锁模型，actor模型对并行也没有提供直接支持。需要通过并发的技术来构造并行的方案，这样就会引入不确定性。而且，由于多个actor并不共享状态，仅通过消息传递来进行交流，所以不太适合实施细粒度的并行。

其他语言

与许多伟大的思想一样，actor模型也由来悠久——20世纪70年代Carl Hewitt首次提出这个模型。Erlang无疑为布道actor做了最大的贡献。比如Erlang的创始人Joe Armstrong也是“任其崩溃”哲学的先驱。

大部分流行的编程语言都提供了一个actor库，特别是Akka库[1]为Java和其他运行于JVM的语言提供了对actor模型的支持。如果想深入学习Akka，建议阅读本书的奖励章节[2]，其中描述了如何用Scala进行actor编程。

---

① http://akka.io

② http://media.pragprog.com/titles/pb7con/Bonus_Chapter.pdf

结语

actor模型是应用最广泛的编程模型之一——不仅提供了并发支持，还支持分布式、错误检测和容错。当面对越来越大的分布式需求时，该模型是解决问题的绝佳选择。

下一章我们将学习通信顺序进程（Communicating Sequential Processes，CSP）。虽然CSP模型看上去类似于actor模型，但区别在于：actor模型的重点在于参与交流的实体，而CSP模型的重点在于用于交流的通道。因此使用CSP模型将是另一番体验。

第 6 章

通信顺序进程

6

如果你和我一样是个车迷，很可能只会关注车辆本身，而忽略了它所要行驶的道路。大家都在喋喋不休地争论涡轮增压与自然吸气孰优孰劣，让中置发动机布局与前置发动机布局一较高下，却忘记了最重要的方面其实与车辆本身无关。你能去往何方、能多快到达目的地，首要的决定因素是道路网络而不是车辆本身。

消息传递系统（message-passing system）与之类似，决定其特性和功能的首要因素并不是用于传递消息的代码或者消息的内容，而是消息的传输通道。

本章我们所考察的模型表面上与actor模型相似，但由于其侧重点不同，所以有着很大的差别。

6.1 万物皆通信

如上一章所述，使用actor模型的程序是由独立的、并发执行的实体（称为actor，Elixir中称为进程）组成的，这些实体之间通过发送消息进行通信。每个actor都有一个信箱，用于保存已经收到但尚未被处理的消息。

与actor模型类似，通信顺序进程（Communicating Sequential Processe，CSP）模型也是由独立的、并发执行的实体所组成，实体之间也是通过发送消息进行通信。但两种模型的重要差别是：CSP模型不关注发送消息的实体，而是关注发送消息时使用的*channel*（通道）。channel是第一类对象，它不像进程那样与信箱是紧耦合的，而是可以单独创建和读写，并在进程之间传递。

与函数式编程和actor模型类似，CSP模型也是正在复兴的古董。由于近来Go语言[①]的兴起，CSP模型又流行起来。我们将通过`core.async`库[②]来介绍CSP模型，这个库将Go的并发模型引入了Clojure。

第一天，我们将学习构建`core.async`库的两大基石：channel和go块。第二天，使用这些知识构建一个有现实意义的例子。第三天，学习如何在ClojureScript中使用`core.async`来简化客户端编程。

① http://golang.org

② http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html

6.2　第一天：channel 和 go 块

core.async提供了两个主要的工具——channel和go块。在大小有限的线程池中，go块允许多个并发任务复用线程资源。现在还是先来看看channel。

使用core.async库

在Clojure语言中，core.async库的资历较浅，且仍处于预发布阶段（因此需要留意可能发生的变化）。要使用这个库，你需要为项目添加依赖并导入core.async。由于core.async库定义的一些函数名与Clojure核心库的函数名冲突，添加依赖和导入库往往较为繁复。为简单起见，你可以使用本书配套代码中的channel项目，它是这样导入core.async库的：

CSP/channels/src/channels/core.clj

```
(ns channels.core
  (:require [clojure.core.async :as async :refer :all
             :exclude [map into reduce merge partition partition-by take]]))
```

通过指定:refer :all，大多数core.async库的函数可以直接使用，但还有一部分（函数名与核心库函数名冲突的函数）必须通过使用async/前缀才能调用。

切换到channel项目下，直接运行lein repl，就可以运行一个REPL，其中已经加载了core.async库的函数定义。

Channel

一个channel就是一个线程安全的队列——任何任务只要持有channel的引用，就可以向一端添加消息，也可以从另一端删除消息。在actor模型中，消息是从指定的一个actor发往指定的另一个actor的；与之不同，使用channel发送消息时发送者并不知道谁是接收者，反之亦然。

通过chan函数可以创建新的channel：

```
channels.core=> (def c (chan))
#'channels.core/c
```

使用>!!可以向channel中写入消息，使用<!!可以从channel中读出消息：

```
channels.core=> (thread (println "Read:" (<!! c) "from c"))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@78fcc563>
channels.core=> (>!! c "Hello thread")
Read: Hello thread from c
nil
```

这段代码使用了core.async提供的thread辅助宏，这个宏会将其中的代码运行在一个单独的线程上。这个线程将会输出从channel中读出的消息。不过首先它会阻塞，直到调用>!!向channel中写入消息，然后我们才会看到输出。

缓存区

默认情况下，channel是同步的（或称无缓存的）——一个任务向channel写入消息的操作会一直阻塞，直到另一个任务从channel中读出消息：

```
channels.core=> (thread (>!! c "Hello") (println "Write completed"))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@78fcc563>
channels.core=> (<!! c)
Write completed
"Hello"
```

如果向chan函数传入缓存区的大小，就可以创建一个有缓存的channel：

```
channels.core=> (def bc (chan 5))
#'channels.core/bc
channels.core=> (>!! bc 0)
nil
channels.core=> (>!! bc 1)
nil
channels.core=> (close! bc)
nil
channels.core=> (<!! bc)
0
channels.core=> (<!! bc)
1
channels.core=> (<!! bc)
nil
```

这段代码创建了一个channel，其缓存区可以容纳五个消息。当channel的缓存区有足够空间时，向其中写入消息的操作会立刻完成，不会阻塞。

关闭channel

上面的代码还展示了channel的另一个特性——可以通过close!关闭channel。从已经关闭的空的channel中读出消息，将得到nil；向已经关闭的channel写入消息，该消息将默默地被弃用。如你所料，向channel中写入nil将发生错误：

```
channels.core=> (>!! (chan) nil)
IllegalArgumentException Can't put nil on channel «...»
```

下面的函数将运用我们已学的知识，不断地从channel中读出消息，直到channel被关闭。函数将以数组形式返回读到的所有内容：

CSP/channels/src/channels/core.clj

```
(defn readall!! [ch]
  (loop [coll []]
```

6

```
    (if-let [x (<!! ch)]
      (recur (conj coll x))
      coll)))
```

在上面的代码中，coll的初始值是空数组[]。每次循环将从ch中读出一个值，如果读出的值不是nil，就将其添加到coll中；如果读出的值是nil（channel已经被关闭），函数将返回coll。

下面是writeall!!函数，其接受一个channel和一个数组，将数组的所有值写入channel，并在写入完成后关闭channel：

CSP/channels/src/channels/core.clj

```
(defn writeall!! [ch coll]
  (doseq [x coll]
    (>!! ch x))
  (close! ch))
```

来测试一下这几个函数：

```
channels.core=> (def ch (chan 10))
#'channels.core/ch
channels.core=> (writeall!! ch (range 0 10))
nil
channels.core=> (readall!! ch)
[0 1 2 3 4 5 6 7 8 9]
```

你肯定料到了core.async会提供具有类似功能的辅助函数，这样我们就不用自己创建这些函数了：

```
channels.core=> (def ch (chan 10))
#'channels.core/ch
channels.core=> (onto-chan ch (range 0 10))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@6b16d3cf>
channels.core=> (<!! (async/into [] ch))
[0 1 2 3 4 5 6 7 8 9]
```

onto-chan函数用于将集合中的所有内容写入channel，并在写入完成时关闭channel。async/into函数接受一个初始集合（上例中是空集合）和一个channel，并返回一个channel。返回的channel中的元素是一个集合，这个集合由初始集合和从输入的channel中读出的所有元素合并而成。

下面我们将使用这些辅助函数进一步讨论有缓存区的channel。

缓存区已满时的策略

默认情况下，向一个缓存区已满的channel中写入消息将会被阻塞。但我们也可以选择其他策略，通过向chan函数传入缓存区来实现：

```
channels.core=> (def dc (chan (dropping-buffer 5)))
#'channels.core/dc
channels.core=> (onto-chan dc (range 0 10))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@147c0def>
```

```
channels.core=> (<!! (async/into [] dc))
[0 1 2 3 4]
```

这段代码创建了一个channel，其使用一个缓存区容量为5的`dropping-buffer`。我们将数字0~9写入channel，虽然channel的缓存区不能容纳这么多数字，但并没有阻塞。如果读出channel中所有的数字，就会发现只有5个数字——后面的数字被弃用了。

Clojure还提供了`sliding-buffer`：

```
channels.core=> (def sc (chan (sliding-buffer 5)))
#'channels.core/sc
channels.core=> (onto-chan sc (range 0 10))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@3071908b>
channels.core=> (<!! (async/into [] sc))
[5 6 7 8 9]
```

与之前一样，这段代码创建了一个容量为5的channel，但这次使用的是`sliding-buffer`。如果读出channel中所有的数字，会发现输出的是最后写入的5个数字——也就是说，向一个缓存区已满的channel中写入数据，`sliding-buffer`将会弃用之前写入的数据。稍后我们还会更详细地研究channel，下面先来看看`core.async`的另一个主要特性——go块。



小乔爱问：

为什么没有容量自动增大的缓存区？

我们已经学习了 `core.async` 库提供的全部三种缓存区类型——阻塞型（blocking）、弃用新值型（dropping）和移出旧值型（sliding）。从感觉上说，如果缓存区能按需增加容量，那也是很合理的。为什么 `core.async` 库没有提供这样的缓存区类型？

其中的原因是个老生常谈的话题，即使你有一个现在看上去“永不枯竭”的资源，总有一天这个资源还是会被用尽。可能是因为时过境迁，当初的程序需要解决更大规模的问题；也可能是因为存在一个 bug，消息没有被及时处理，从而导致堆积。

如果你放弃思考相应的对策，那未来的某个时间就有可能出现一个破坏性极强、隐蔽极深且难以诊断的 bug。实际上，让进程的信箱溢出，是让 Erlang 系统全面崩溃的为数不多的方法之一[a]。最好的策略是在现在就思考如何处理缓存区被塞满的情况，将问题消灭在萌芽状态。

a. http://prog21.dadgum.com/43.html

go块

线程启动和运行时都有一定开销，这正是现在的程序都避免直接创建线程、转而使用线程池

（参见2.4节的“创建线程之终极版”部分）的原因。实际上，我们在以前的例子中见过的`thread`宏内部也使用了`CachedThreadPool`。

然而线程池并不总是适用。尤其是当程序阻塞时，使用线程池可能会造成麻烦。

阻塞带来的问题

线程池技术是处理CPU密集型任务的利器——任务进行时会占用某个线程，任务结束后将线程返还给线程池，使线程可以被复用。但涉及线程通信时使用线程池是否仍然合适呢？如果线程被阻塞，那么它将无限期被占用，这就削弱了使用线程池技术的优势。

这种问题是有一些解决方案的，但它们通常会对代码风格加以限制，使之变成事件驱动的形式。事件驱动是一种编程风格，对于从事UI编程或事件类服务器编程的程序员来说一定不陌生。

虽然这些方案都能解决问题，但它们破坏了控制流的自然的表达形式，让代码变得难以阅读和理解。更糟糕的是，这些方案还会大量使用全局状态，因为事件处理器需要保存一些数据，以便之后的事件处理器使用。我们已经学习过这个结论：状态和并发最好不要混用。

go块提供了一种两全其美的解决方案——既可以写出事件驱动的代码来解决目前碰到的阻塞问题，又可以不牺牲代码的结构性和可读性。其原理是go块在底层将串行化代码透明地重写成了事件驱动的形式。

控制反转

与其他Lisp方言类似，Clojure有一套强大的宏系统。如果你用过其他语言的宏系统（比如C/C++中的预处理器宏），就会觉得Lisp的宏系统更像是魔法，它可以进行神奇的代码变换。`go`宏就是其中一个小魔法。

`go`块中的代码会被转换成一个状态机。当从channel中读出消息或向channel中写入消息时，状态机将暂停，并释放它所占用的线程的控制权。当代码可以继续运行时，状态机进行一次状态转换，并可能在另一个线程中继续运行。

通过这样的控制反转，`core.async`运行时可以在有限的线程池中高效地运行许多go块。我们先来看一个例子，稍后再看看到底有多么高效。

状态机暂停

下面是使用go块的一个例子：

```
channels.core=> (def ch (chan))
#'channels.core/ch
channels.core=> (go
           #_=>   (let [x (<! ch)
           #_=>         y (<! ch)]
           #_=>     (println "Sum:" (+ x y))))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@13ac7b98>
channels.core=> (>!! ch 3)
```

```
nil
channels.core=> (>!! ch 4)
nil
Sum: 7
```

这段代码首先创建了一个channel `ch`。然后创建了一个go块，用来从`ch`中读取两个值，并输出两个值的和。虽然看上去go块从channel中读取数据时应当阻塞，实际上却发生了有趣的事情。

这段代码并没有用`<!!`从channel中读取数据，而是使用了`<!`。单个叹号意味着本次读channel是进行暂停操作，而不是进行阻塞操作。同理，`>!`是`>!!`的暂停版本[①]。

如图6-1所示，go块将串行的代码转换成有3个状态的状态机。

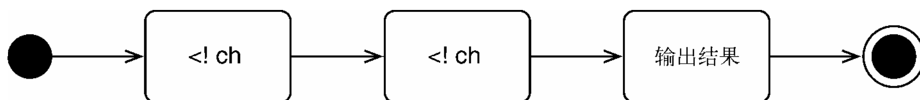


图6-1　串行代码对应的状态机

该状态机包括以下3个状态：

(1) 初始状态会直接暂停，等待`ch`中有数据可以被读取。满足条件时，状态机进入状态2。

(2) 状态机首先将从`ch`中读取的值绑定到`x`上，然后暂停，等待`ch`中下一个可以被读取的数据。满足条件时，状态机进入状态3。

(3) 状态机将从`ch`中读取的值绑定到`y`上，输出结果，并终止。



小乔爱问：

如果go块中发生阻塞呢？

如果 go 块中使用了一个阻塞函数，比如`<!!`，那么当前运行的线程会被阻塞。虽然代码的正确性不会受到影响（不过，如果阻塞了足够多的线程，会因为没有可运行的线程而陷入死锁），但是这样做违背了使用 go 块的本意。如果误用了阻塞函数，你不会看到警告，也就是说你要保证使用了正确的函数。

幸运的是，如果在不能使用暂停函数的地方使用了暂停函数，你会得到警告：

```
channels.core=> (<! ch)
AssertionError Assert failed: <! used not in (go ...) block
nil  clojure.core.async/<! (async.clj:83)
```

① 之后将“进行暂停操作的函数”简称为“暂停函数”；将“进行阻塞操作的函数”简称为“阻塞函数”
——译者注

go块的成本很低

go块的意义主要在于其效率。与使用线程不同，使用go块的成本很低，因此可以创建很多go块而不用担心耗尽资源。这看上去是个小小的改进，但实际上，不用担心资源而能随意创建并发任务有着革命性的意义。

你也许注意到了go返回的是一个channel（thread也是返回channel）。go块运行完成时会将结果写到这个channel中：

```
channels.core=> (<!! (go (+ 3 4)))
7
```

下面这个简单的函数会创建大量go块，从结果可以看出go块的成本是很低的：

CSP/channels/src/channels/core.clj

```
(defn go-add [x y]
  (<!! (nth (iterate #(go (inc (<! %))) (go x)) y)))
```

这个函数可能是“世界上最低效的加和函数”了。它创建了y个go块形成的流水线，其中每一个go块都将其参数加1。

来分析一下其工作过程的每个阶段：

(1) 匿名函数`#(go (inc (<! %)))`创建了一个go块，这个go块接受一个channel，从中读出一个值，并返回一个channel（其中包含了递增后的值）；
(2) 上述匿名函数被传给`iterate`，`iterate`使用的初始值是`(go x)`（这个channel中只包含`x`）。回忆一下，`iterate`会返回如下形式的懒惰数组：`(x (f x) (f (f x)) (f (f (f x))) ...)`；
(3) 使用`nth`读出上述数组中第y个元素，这是一个channel，其中的值是将x递增y次的结果；
(4) 使用`<!!`从上述channel中读出结果。

来测试一下这段代码：

```
channels.core=> (time (go-add 10 10))
"Elapsed time: 1.935 msecs"
20
channels.core=> (time (go-add 10 1000))
"Elapsed time: 5.311 msecs"
1010
channels.core=> (time (go-add 10 100000))
"Elapsed time: 734.91 msecs"
100010
```

可以看到，创建并运行100 000个go块需要花费3/4秒。这意味着go块的性能比起Elixir的进程毫不逊色——这个成绩非常优秀，因为Elixir运行在以并发性能为设计主旨的Erlang虚拟机中，而Clojure却是运行在JVM中。

我们已经学习了channel和go块这两种技术，现在可以将两者结合使用，构造出更复杂的channel操作。

在channel上进行操作

如果你感觉channel与数组有些相像，那你并没有错。与数组类似，channel代表了一系列有序的值；我们可以将一些高级函数施加在channel中的全部元素上——比如map函数、filter函数等；我们还可以将这些函数串联起来，构建复杂的操作。

在channel上进行映射

下面是channel版的map函数：

CSP/channels/src/channels/core.clj

```
(defn map-chan [f from]
  (let [to (chan)]
    (go-loop []
      (when-let [x (<! from)]
        (>! to (f x))
        (recur))
      (close! to))
    to))
```

这个函数接受一个函数f和一个源channel from。首先，这段代码创建了一个目标channel to，to将作为函数的返回值。然后，使用go-loop创建一个go块，go-loop是一个辅助函数，等价于(go (loop ...))。循环体中使用when-let从from中读出值并绑定到x上。如果x不为null，则when-let中的代码会被执行，(f x)将被写入to中，并且循环会继续进行。如果x为null，to会被关闭。

测试一下这个函数：

```
channels.core=> (def ch (chan 10))
#'channels.core/ch
channels.core=> (def mapped (map-chan (partial * 2) ch))
#'channels.core/mapped
channels.core=> (onto-chan ch (range 0 10))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@9f3d43e>
channels.core=> (<!! (async/into [] mapped))
[0 2 4 6 8 10 12 14 16 18]
```

按照惯例，core.async提供了类似于map-chan的函数map<。它还提供了filter的channel版filter<、mapcat的channel版mapcat<，等等。我们还可以组合使用这些函数，串联成一个channel的处理链：

```
channels.core=> (def ch (to-chan (range 0 10)))
#'channels.core/ch
channels.core=> (<!! (async/into [] (map< (partial * 2) (filter< even? ch))))
[0 4 8 12 16]
```

上面这段代码使用了`core.async`提供的另一个辅助函数`to-chan`，其创建并返回一个channel，这个channel中包含了输入数组中的所有元素，在数组中的元素用尽后这个channel会关闭。

现在来做一个有趣的试验，以此为第一天的学习做个结尾。

并发版本的埃氏筛

我们现在来实现一个并发版本的埃氏筛[1]。`get-primes`函数会返回一个channel，其中包含了`limit`以内（含`limit`）的所有素数（从小到大排列）：

CSP/Sieve/src/sieve/core.clj

```
(defn factor? [x y]
  (zero? (mod y x)))

(defn get-primes [limit]
  (let [primes (chan)
        numbers (to-chan (range 2 limit))]
    (go-loop [ch numbers]
      (when-let [prime (<! ch)]
        (>! primes prime)
        (recur (remove< (partial factor? prime) ch)))
      (close! primes))
    primes))
```

稍后将简要介绍这段代码的工作原理（建议你先自己整理一下思路——所有必要的知识之前都已经介绍过了）。我们还是先来验证一下其正确性。下面的`main`函数会调用`get-primes`，并输出其返回的channel中的所有值：

CSP/Sieve/src/sieve/core.clj

```
(defn -main [limit]
  (let [primes (get-primes (edn/read-string limit))]
    (loop []
      (when-let [prime (<!! primes)]
        (println prime)
        (recur)))))
```

运行一下，会得到以下结果：

```
$ lein run 100000
2
3
5
7
11
⋮
99971
99989
99991
```

① 埃拉托斯特尼（Eratosthenes）筛法，简称埃氏筛，是一种检定素数的简易算法。——译者注

现在来介绍`get-primes`的工作原理。首先，创建一个channel `primes`，`primes`将作为函数的返回值。然后，进入循环，`ch`的初始值是`numbers`，`to-chan`负责将从2到`limit`之间的整数写入`numbers`。

首先，循环从`ch`中读取第一个元素，这个元素肯定是素数（之后会解释原因），所以将被写入`primes`。然后，进入下一轮循环，与第一轮循环不同的是`ch`的值变为`(remove< (partial factor? prime) ch)`的结果。

`remove<`函数类似于`filter<`，区别在于它返回一个channel，其中只包含断言为`false`的值。在本例中，这排除了以上一轮认定的素数为因子的所有数。

综上所述，`get-primes`创建了一个channel的流水线。第一个channel包含从2到`limit`的所有整数，第二个channel排除了以2为因子的所有整数，第三个channel排除了以3为因子的所有整数，以此类推（如图6-2所示）。

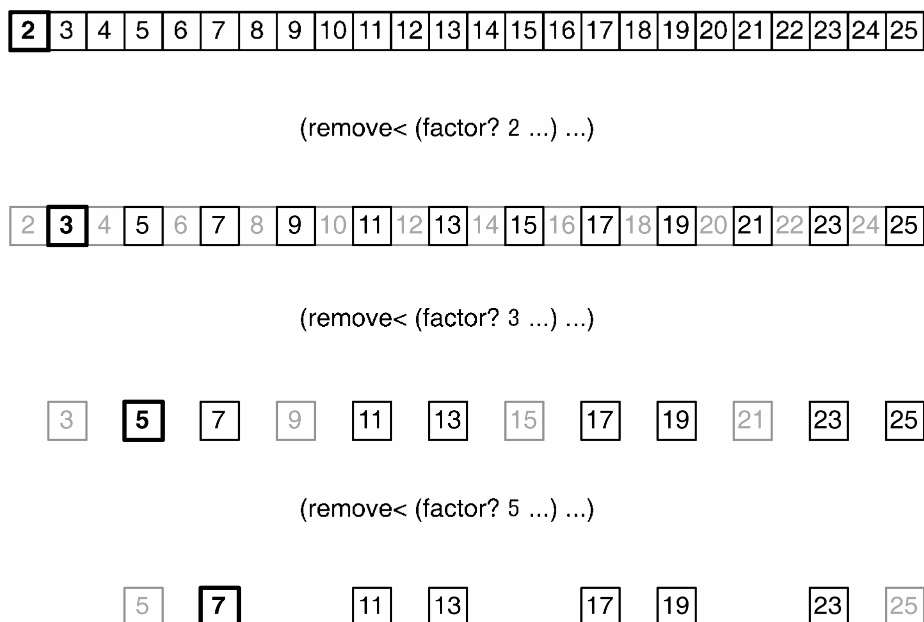


图6-2　并发版本的埃氏筛

希望大家不要误会，上面这个例子并不是实现并行埃氏筛的最佳方法——它为了演示功能而滥用了channel。不过这很好地演示了如何将channel组合在一起实现某种特定的通信模式。

第一天总结

第一天的学习即将结束。第二天将学习如何从多个channel中读出数据，以及如何用channel和go块构造IO密集型的程序。

第一天我们学到了什么

`core.async`的两大基石是channel和go块：

- 默认情况下，channel是同步的（无缓存的）——向一个channel写入数据的操作将一直被阻塞，直到另一个任务从channel中读出消息；
- channel也可以是有缓存的。在缓存区已满时可以使用不同的缓存策略——阻塞型（blocking）、弃用新值型（dropping）和移出旧值型（sliding）；
- 通过控制反转，go块将串行代码重写成一个状态机。go块不会进行阻塞，而是暂停状态机，这样当前所处的线程就可以为另一个go块所使用；
- channel操作的阻塞版本的函数名是以两个感叹号（`!!`）结尾，而暂停版本的函数名是以一个感叹号（`!`）结尾。

第一天自习

查找

- 阅读`core.async`的官方文档。
- 观看Timothy Baldridge的视频教程"Core Async Go Macro Internals"，或阅读Huey Petersen的博文"The State Machines of core.async"。这两篇文献都描述了`go`宏是如何实现控制反转的。

实践

- 本节的`map-chan`创建并返回了一个同步的（无缓存的）channel。如果其使用一个有缓存的channel会发生什么？哪一种选择更好？什么情况下适用有缓存的channel？
- `core.async`不仅提供了`map<`，还提供了`map>`。这两者有什么区别？尝试自己实现一个`map>`。`map<`和`map>`分别适用于什么场景？
- 实现一个基于channel的并行map函数（类似于Clojure中的`pmap`函数，或者类似于在前面章节中用Elixir实现的并行map函数）。

6.3　第二天：多个 channel 与 IO

今天将学习如何使用`core.async`库，让异步IO的处理变得简洁易懂。在此之前，需要先了解一个之前没有提及的特性——如何同时处理多个channel。

处理多个channel

到目前为止，我们在某个时间仅处理一个channel，但我们能做的并不仅限于此。使用`alt!`宏可以处理多个channel：

```
channels.core=> (def ch1 (chan))
#'channels.core/ch1
channels.core=> (def ch2 (chan))
#'channels.core/ch2
channels.core=> (go-loop []
           #_=>   (alt!
           #_=>     ch1 ([x] (println "Read" x "from channel 1"))
           #_=>     ch2 ([x] (println "Twice" x "is" (* x 2))))
           #_=>   (recur))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@d8fd215>
channels.core=> (>!! ch1 "foo")
Read foo from channel 1
nil
channels.core=> (>!! ch2 21)
Twice 21 is 42
nil
```

这段代码中，首先创建了两个channel：`ch1`和`ch2`，然后创建了一个go块，其会不断循环并使用`alt!`从两个channel中读取数据。如果能从`ch1`中读取数据，那么将数据直接输出；如果能从`ch2`中读取数据，那么将数据翻倍后再输出。

从这段代码中很容易就能理解`alt!`宏的工作原理——它接受成对的参数，每对的第一个参数是一个channel，第二个参数是一段代码，从channel中读出数据后将执行这段代码。在本例中，这段代码看上去像是一个匿名函数，从channel中读出的值被赋给`x`，并通过`println`输出。但它实质上并不是匿名函数——它没有使用`fn`构造匿名函数。

这就是Clojure的宏系统施加的另一个魔法，比起使用匿名函数，`alt!`宏的这种用法显得更简洁高效。



小乔爱问：

如果是向多个channel中写入数据呢？

我们刚才只是学习了`alt!`宏的皮毛——类似于从多个channel读出数据，`alt!`宏也可以用于向多个channel写入数据，或者将读写混用。本书并不会涉及这种用法，如果读者想深入了解`alt!`，可以查看官方文档。

超时

`timeout`函数返回一个channel，这个channel在指定的时间（以毫秒为单位）过后会被关闭：

```
channels.core=> (time (<!! (timeout 10000)))
"Elapsed time: 10001.662 msecs"
nil
```

`timeout`与`alt!`一起使用，可以为channel操作设置超时时间，比如：

```
channels.core=> (def ch (chan))
#'channels.core/ch
channels.core=> (let [t (timeout 10000)]
           #_=> (go (alt!
           #_=>   ch ([x] (println "Read" x "from channel"))
           #_=>   t (println "Timed out"))))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@28134be9>
channels.core=>
Timed out
```

设置超时并没有什么新鲜，但这种方法使得超时具象化了（用一个对象来代表超时操作），稍后将会看到这项改进是非常有用的。

具象化的超时

大部分系统是以请求为对象来设置超时时间的，比如Java的`URLConnection`类提供的`setReadTimeout()`方法。如果服务器在一定时间内没有响应，`read()`会抛出异常`IOException`。

这只适用于对单个请求设置超时时间，如果要为几个串行的请求设置一个总的超时时间，上述方法就不能解决问题，而具象化的超时却可以。只需要创建一个超时对象，并在串行的几个请求中都使用这个超时对象即可。

为说明这一点，我们将昨天创建的素数筛稍微修改一下，使其不接受数值范围，而是接受一个时间。素数筛将在规定时间内尽可能多地产生素数。

先修改`get-primes`，使其不断产生素数：

CSP/SieveTimeout/src/sieve/core.clj

```
(defn get-primes []
  (let [primes (chan)
➤       numbers (to-chan (iterate inc 2))]
    (go-loop [ch numbers]
      (when-let [prime (<! ch)]
        (>! primes prime)
        (recur (remove< (partial factor? prime) ch)))
      (close! primes))
    primes))
```

之前channel的初始值是由`(range 2 limit)`产生的，而这次是用无穷数组`(iterate inc 2)`。

下面的代码使用了`get-primes`：

CSP/SieveTimeout/src/sieve/core.clj

```
(defn -main [seconds]
```

```
  (let [primes (get-primes)
➤       limit (timeout (* (edn/read-string seconds) 1000))]
    (loop []
➤     (alt!! :priority true
➤       limit nil
➤       primes ([prime] (println prime) (recur))))))
```

这段代码中使用了`alt!!`，如你所料，它是`alt!`的阻塞版本。这段代码中的`alt!!`将进行阻塞，直到产生了一个新的素数，或者到达设置的超时时间`limit`而返回`nil`。`:priority true`选项确保了`alt!!`的子句（都可以执行时）是按代码顺序执行的（默认情况下，如果两个子句都可以执行，它们执行的顺序是不确定的），这样就尽量避免了由于产生素数的速度过快而导致超时的子句无法执行的情况。这个例子展示了如何自然地设置超时时间——相比为每个请求设置超时时间，这种方式显得更加自然。

下一节将使用超时机制和Clojure的宏系统来构建一个辅助工具，它适用于一个普遍的场景——轮询。

异步轮询

稍后我们将构建一个RSS阅读器。RSS阅读器需要轮询指定的新闻feed来检测是否有新的文章。本节我们将使用超时机制和Clojure的宏系统来构建一个辅助工具，它可以轻松高效地进行异步轮询。

轮询函数

要实现轮询功能，需要用到之前提到的`timeout`函数。下面这个函数接受两个参数：轮询周期（以秒为单位）和一个函数，这个函数每隔一个轮询周期会被调用一次：

CSP/Polling/src/polling/core.clj

```
(defn poll-fn [interval action]
  (let [seconds (* interval 1000)]
    (go (while true
          (action)
          (<! (timeout seconds))))))
```

这个函数十分简单，并且能正常运行：

```
polling.core=> (poll-fn 10 #(println "Polling at:" (System/currentTimeMillis)))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@6e624159>
polling.core=>
Polling at: 1388827086165
Polling at: 1388827096166
Polling at: 1388827106168
⋮
```

不过这里有一个问题——也许你看到`poll-fn`是在go块中对传入的函数进行调用的，因此就认为传入的函数中可以调用暂停函数。如果这样做，会发生以下异常：

```
polling.core=> (def ch (to-chan (iterate inc 0)))
#'polling.core/ch
polling.core=> (poll-fn 10 #(println "Read:" (<! ch)))
Exception in thread "async-dispatch-1" java.lang.AssertionError:
  Assert failed: <! used not in (go ...) block
nil
```

问题的原因在于暂停函数必须直接在go块中调用——在这一点上Clojure的宏魔法也无能为力。

轮询宏

之前使用函数进行轮询，而另一种轮询的方法是使用宏：

CSP/Polling/src/polling/core.clj

```
(defmacro poll [interval & body]
  `(let [seconds# (* ~interval 1000)]
     (go (while true
           (do ~@body)
           (<! (timeout seconds#))))))
```

本书不会详细介绍Clojure的宏，因此你只能暂且接受本节的内容。稍后会详细讲解`poll`的工作原理，下面这几点会帮助我们理解其工作原理。

- 编译系统不是直接编译宏，而是编译宏展开后的代码。
- 反引号（`）是一个运算符，用于引用代码。它并不执行其中的代码，而是返回代码的可编译形式。
- 在宏中，可以通过~和~@操作符来引用宏的参数。
- 在变量名后使用#后缀，可以让Clojure自动生成一个唯一名字（以确保宏使用的变量名和传给宏的代码中的变量名不会冲突）。

来测试一个宏`poll`：

```
polling.core=> (poll 10
          #_=>   (println "Polling at:" (System/currentTimeMillis))
          #_=>   (println (<! ch)))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@1bec079e>
polling.core=>
Polling at: 1388829368011
0
Polling at: 1388829378018
1
⋮
```

由于宏是在编译时展开，因此传给`poll`的代码是内联的（inlined），会被直接替换到`poll`的go块中，也就是说代码中可以包含暂停函数。使用宏的好处不只如此。我们不必再传入一个完整的函数，而是传入一个代码片段，这样就不用再借助匿名函数，进而代码看上去会更自然。事实上，我们创建了一种（不同于函数的）控制结构。

通过展开poll可以检查它是否正确：

```
polling.core=> (macroexpand-1
         #_=>   '(poll 10
         #_=>      (println "Polling at:" (System/currentTimeMillis))
         #_=>      (println (<! ch))))
(clojure.core/let [seconds__2691__auto__ (clojure.core/* 10 1000)]
  (clojure.core.async/go
    (clojure.core/while true
      (do
        (println "Polling at:" (System/currentTimeMillis))
        (println (<! ch)))
      (clojure.core.async/<! (clojure.core.async/timeout seconds__2691__auto__)))))
```

为了方便阅读，本书对上面代码中macroexpand-1的输出结果进行了格式上的调整。可以看到，传给poll的代码被“粘贴”到了宏的代码中，并且seconds#被转换成了一个唯一名字（如果传给poll的代码中，seconds这个名字有其他用途，那么这个转换就尤为必要了）。

下一节的例子中会使用poll宏。

异步IO

在IO这个领域，异步代码身手不凡——与传统的一个线程进行一个连接不同，异步IO可以一下进行很多连接，当其中有一个连接的数据可用时，会收到一个通知。这是一个很强大的技术，但使用起来也颇具挑战性，因为异步代码往往会是回调嵌套回调，慢慢演变成一团糟。本节我们来学习core.async是如何让异步代码变得简洁的。

继续前几章词频统计的例子，本节将创建一个RSS阅读器，用来监听新闻feed，当检测到新的文章时进行词数统计[①]。我们将创建若干并发的go块，并将其用channel连接成一个流水线：

(1) 底层的go块用于监听某个feed，每60秒进行一次轮询。它首先解析轮询得到的XML，然后从中提取出文章的链接，最后将其传给流水线。
(2) 下一层的go块维护了从某个feed中收到的文章链接的列表。当它发现一篇新的文章时，就将文章的链接传给流水线。
(3) 下一层的go块依次接受新的文章链接，并统计文章中的词数，再将计数结果传给流水线。
(4) 将多个feed的计数结果合并到一个channel中。
(5) 最高层的go块监听合并后的channel，当有新的计数结果产生时，将该结果输出。

整个流水线的结构如图6-3所示。

6

① 与前几章不同，此处只统计词数，而不统计词频。——译者注

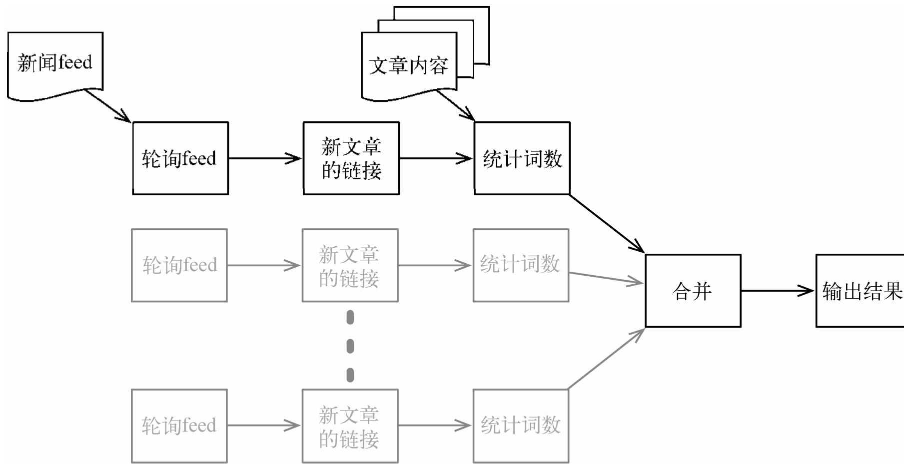


图6-3　RSS阅读器的结构

先来看看如何将一个现成的异步IO库集成到`core.async`中。

从回调转向channel

本例将使用http-kit库[①]。与许多异步IO库类似，当一个操作完成时，http-kit会调用回调函数：

```
wordcount.core=> (require '[org.httpkit.client :as http])
nil
wordcount.core=> (defn handle-response [response]
            #_=>   (let [url (get-in response [:opts :url])
            #_=>         status (:status response)]
            #_=>     (println "Fetched:" url "with status:" status)))
#'wordcount.core/handle-response
wordcount.core=> (http/get "http://paulbutcher.com/" handle-response)
#<core$promise$reify__6310@3a9280d0: :pending>
wordcount.core=>
Fetched: http://paulbutcher.com/ with status: 200
```

现在的任务是对`http/get`进行封装，将http-kit集成到`core.async`中。这里将用到一个陌生的函数`put!`，它不必在go块中调用。它可以向channel写入数据，与之前不同，它只是发起写入而并不关心结果（既不会进行阻塞也不会进行暂停）：

CSP/WordCount/src/wordcount/http.clj

```
(defn http-get [url]
  (let [ch (chan)]
    (http/get url (fn [response]
                    (if (= 200 (:status response))
                      (put! ch response)
                      (do (report-error response) (close! ch)))))
    ch))
```

① http://http-kit.org

这段代码首先创建了一个channel，它将作为函数的返回值（你应该已经很熟悉这个模式了）；然后调用了`http/get`，并立刻返回。在未来某个时间，当GET操作完成时，回调函数将被调用。如果GET操作返回的状态是200（表示成功），回调函数会将GET操作的响应内容写入channel；如果状态不是200，回调函数会报告一个错误并关闭channel。

下一节将创建轮询RSS feed的函数。

轮询feed

我们已经有了`http-get`和`poll`，如你所料，通过简单的代码就可以轮询RSS feed：

CSP/WordCount/src/wordcount/feed.clj

```
(def poll-interval 60)

; Simple-minded feed-polling function
; WARNING: Don't use in production (use conditional get instead)

(defn poll-feed [url]
  (let [ch (chan)]
    (poll poll-interval
      (when-let [response (<! (http-get url))]
        (let [feed (parse-feed (:body response))]
          (onto-chan ch (get-links feed) false))))
    ch))
```

`parse-feed`和`get-links`这两个函数会使用Rome库[1]来解析feed所返回的XML。本书不再介绍这两个函数，你可以在本书配套代码中找到它们。

`get-links`会返回链接的列表，通过`onto-chan`将这个列表写入`ch`。默认情况下，`onto-chan`会在写完列表后关闭channel，这里通过设置`onto-chan`的最后一个参数使其不关闭channel。

6

来测试一下`poll-feed`：

```
wordcount.core=> (ns wordcount.feed)
nil
wordcount.feed=> (def feed (poll-feed "http://www.cbsnews.com/feeds/rss/main.rss"))
#'wordcount.feed/feed
wordcount.feed=> (loop []
            #_=>   (when-let [url (<!! feed)]
            #_=>     (println url)
            #_=>     (recur)))
http://www.cbsnews.com/news/three-year-old-dies-after-visit-to-dentist-in-hawaii/
http://www.cbsnews.com/news/obama-unemployment-benefits-expiration-just-plain-cruel/
http://www.cbsnews.com/news/rand-paul-says-hes-suing-over-nsa-surveillance-programs/
 ⋮
```

① http://rometools.github.io/rome/

下面来看看如何对poll-feed所返回的链接进行去重。

请勿轻易尝试

对于本书来说，用这样简单的轮询策略来举例是为了方便理解，请不要在产品环境中使用这个策略。轮询时每次都获取完整的feed是不必要的，这会增加网络带宽的压力和被轮询服务器的压力，可以通过HTTP的条件GET[a]来减小压力。

a. http://fishbowl.pastiche.org/2002/10/21/http_conditional_get_for_rss_hackers/

对链接进行去重

poll-feed函数进行轮询时，会返回feed中所有的链接，其中包含大量重复的链接。我们需要的channel应该只包含feed中出现的新链接。可以用下面这个函数达到目的：

CSP/WordCount/src/wordcount/feed.clj

```
(defn new-links [url]
  (let [in (poll-feed url)
        out (chan)]
    (go-loop [links #{}]
      (let [link (<! in)]
        (if (contains? links link)
          (recur links)
          (do
            (>! out link)
            (recur (conj links link))))))
    out))
```

首先，这段代码创建了两个channel：in和out。in是由poll-feed函数返回的；feed中的新链接将被写入out。这段代码在go块中进行了一个循环，用于维护目前接收到的链接的集合links，links的初始值是空集合#{}。每当从in中读出一个链接时，就需要检查links中是否已存在该链接。如果已经存在，就什么也不做；否则就将其写入out并添加到links中。

在REPL中测试一下，这次不再会每隔60秒输出一堆链接，而是仅在检测到新链接时才会有输出。

现在我们已经从feed中获取了新文章的链接，接下来就可以依次获取文章的内容并统计词数了。

统计词数

根据之前所学的知识，很容易就可以实现get-counts函数：

CSP/WordCount/src/wordcount/core.clj

```
(defn get-counts [urls]
  (let [counts (chan)]
```

```
(go (while true
      (let [url (<! urls)]
        (when-let [response (<! (http-get url))]
          (let [c (count (get-words (:body response)))]
            (>! counts [url c]))))))
counts))
```

这段代码接受一个channel urls，对于从中读出的每一个链接，使用http-get来获取文章的内容，统计其中的词数，并将结果写入返回的channel中。统计词数的结果是一个二元数组，其中第一个元素是文章的链接，第二个元素是文章的词数。

万事俱备，只差将各个部分组装起来。

组装

下面这个main函数实现了完整的RSS词数统计功能：

CSP/WordCount/src/wordcount/core.clj

```
Line 1 (defn -main [feeds-file]
   2     (with-open [rdr (io/reader feeds-file)]
   3       (let [feed-urls (line-seq rdr)
   4             article-urls (doall (map new-links feed-urls))
   5             article-counts (doall (map get-counts article-urls))
   6             counts (async/merge article-counts)]
   7         (while true
   8           (println (<!! counts))))))
```

这个函数接受一个文件名作为参数，该文件包含了多个feed的URL，每行一个URL。首先，这段代码创建了一个文件读取器（第2行）（Clojure中的with-open函数确保当代码运行到其作用范围之外时，文件会被关闭）；然后，通过line-seq（第3行）从文件读取器中获取URL的列表，并对URL列表施加映射操作（映射函数是new-links）以将其转换为channel的序列（第4行），当检测到某个feed中有新的链接时，新的链接就会被写入对应的channel中；接下来，对channel的序列施加映射操作（映射函数是get-counts）以得到另一个channel的序列（第5行），当检测到某个feed中有新的链接时，链接指向的文章的词数就会被写入这个序列中对应的channel中。

最后，使用async/merge函数（第6行）来将这个channel序列合并为一个单独的channel，其包含了原始channel序列中的所有内容。代码会一直循环下去（第7行），输出所有写入合并后的channel中的词数统计结果。来测试一下：

```
$ lein run feeds.txt
[http://www.bbc.co.uk/sport/0/football/25611509 10671]
[http://www.wired.co.uk/news/archive/2014-01/04/time-travel 11188]
[http://news.sky.com/story/1190148 3488]
⋮
```

在这段程序运行时监测一下CPU的使用情况，会发现这段代码不仅简单易读，而且非常高效，它可以同时检测数百个feed，但只占用少量CPU资源。



小乔爱问：

为什么使用无缓存的channel?

回顾一下今天创建的所有 channel——它们全都是无缓存的（同步的）。学习 CSP 模型的新手往往会认为有缓存的 channel 会比无缓存的 channel 应用更为广泛，但实际情况恰恰相反。有一些场景适合使用有缓存的 channel，但在使用前务必深思熟虑，一定要确认使用缓存的必要性。

第二天总结

我们完成了第二天的学习。第三天将学习在客户端如何通过ClojureScript使用`core.async`库。

第二天我们学到了什么

使用channel和go块，可以写出高效的异步代码，同时代码的表达也非常自然，而不像使用回调函数时那样晦涩。

- 如果要将现存的基于回调机制的API迁移到CSP机制中，只需提供一个很小的回调函数，向channel写入数据即可。
- 通过`alt!`宏可以处理多个channel的读和写。
- `timeout`函数返回一个channel，并在一定时间后关闭这个channel——这样使得“超时”这个操作变成了第一类对象（被具象化了）。
- 暂停函数必须在go块中被直接调用。Clojure的宏可以将代码内联，可以将go块拆分成较小的部分，而不受这个约束的影响。

第二天自习

查找

- 与`alt!`类似，`core.async`还提供了`alts!`。两者有什么区别？各自适用于什么场景？
- 除了`async/merge`，`core.async`还提供了很多方法来合并多个channel，例如`pub`、`sub`、`mult`、`tap`、`mix`和`admix`。它们各适用于什么场景？

实践

- 重新整理一下RSS阅读器运行的流程。有趣的是由于全程使用了无缓存的channel，整个流程看上去非常像数据流式编程的结果，其中上游的go块的运行结果由下游的go块使用。如果使用有缓存的channel会发生什么？比起使用无缓存的channel，是否有什么好处？会带来什么问题？

- ❑ 自己实现一个类似于`async/merge`的函数。这个函数还需要处理输入channel中有一个或多个被关闭的情况。（提示：比起`alt!`，借助`alts!`来实现会比较容易。）
- ❑ 使用Clojure的宏展开工具将`alt!`宏展开：

```
channels.core=> (macroexpand-1 '(alt! ch1 ([x] (println x)) ch2 ([y] (println y))))
```

对于展开后的代码，通过调整缩进并删除`clojure.core`前缀，可以方便阅读。`alt!`是如何不调用匿名函数而达到调用匿名函数的效果的？

6.4 第三天：客户端 CSP

ClojureScript（http://clojurescript.com）是Clojure的一个子集，ClojureScript并不将程序编译成Java字节码，而编译成JavaScript。也就是说可以用Clojure为一个Web应用同时编写服务器端和客户端的代码。

使用ClojureScript的主要原因之一是它支持`core.async`，这为我们带来了许多好处，其中最重要的就是它能将众多JavaScript程序员从“回调困境”中解救出来。

并发是一种心境

如果你熟悉客户端JavaScript编程，肯定会认为本节写错了——浏览器使用的JavaScript引擎是单线程的，怎么会与`core.async`扯上关系？并发编程不是在多线程的场景下才有用吗？

在没有真正多线程的场景中，通过`go`宏的控制反转，ClojureScript可以让客户端编程在表面上具有多线程功能。这是协作式多任务（cooperative multitasking）的一种形式——一个任务不会强制打断另一个任务。之后我们会看到，这为代码结构和清晰度带来了质的飞跃。



小乔爱问：

关于Web Worker

通过 Web Worker[a]，现代浏览器在一定形式上支持真正多线程的 JavaScript。Web Worker 只涉及后台任务，而不能访问 DOM。

通过某些库，比如 Servant[b]，ClojureScript 也可以使用 Web Worker。

a. http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html
b. https://github.com/MarcoPolo/servant

Hello, ClojureScript

ClojureScript与Clojure类似，但也存在一些差别——本书会在相关部分介绍它们。

ClojureScript应用的编译过程通常是两阶段的。第一阶段，客户端的ClojureScript代码将被编译成一个JavaScript文件；第二阶段，服务器端的ClojureScript代码将被编译并创建一个服务器，这个服务器提供的页面会将该JavaScript文件的代码包装在`<script>`标签对中。本节中的例子都使用Leiningen的插件lein-cljsbuild[①]将编译过程自动化。服务器端的代码放在目录`src-clj`中，客户端的代码放在目录`src-cljs`中。

先来看一个简单的项目，将一个脚本嵌入一个页面中，页面如下：

CSP/HelloClojureScript/resources/public/index.html

```
Line 1 <html>
     2   <head>
     3     <title>Hello ClojureScript</title>
     4     <script src="/js/main.js" type="text/javascript"></script>
     5   </head>
     6   <body>
     7     <div id="content">
     8     </div>
     9   </body>
    10 </html>
```

第4行中引用了ClojureScript生成的JavaScript脚本，它将操作第7行空白的`<div>`。下面是对应的ClojureScript代码：

CSP/HelloClojureScript/src-cljs/hello_clojurescript/core.cljs

```
Line 1 (ns hello-clojurescript.core
     -   (:require-macros [cljs.core.async.macros :refer [go]])
     -   (:require [goog.dom :as dom]
     -             [cljs.core.async :refer [<! timeout]]))
     5
     - (defn output [elem message]
     -   (dom/append elem message (dom/createDom "br")))
     - (defn start []
     -   (let [content (dom/getElement "content")]
    10     (go
     -       (while true
     -         (<! (timeout 1000))
     -         (output content "Hello from task 1")))
     -     (go
    15       (while true
     -         (<! (timeout 1500))
     -         (output content "Hello from task 2")))))
     -
     - (set! (.-onload js/window) start)
```

① https://github.com/emezeske/lein-cljsbuild

ClojureScript与Clojure的一个差别是其使用的宏需要使用`:require-macros`进行声明（第2行）。`output`函数（第6行）使用了Google Closure库[①]（此处的Closure第四个字母是s，而不是j）向一个DOM元素添加消息。

这段代码在第13行和第17行使用了`output`函数，其分别运行于两个独立的go块中。第一个go块每1秒输出一次，第二个go块每1.5秒输出一次。

第19行的代码将`start`函数与JavaScript的window对象的`onload`属性进行绑定。这行代码借助了ClojureScript的dot special form特性，与JavaScript代码进行交互。下面这段代码：

```
(set! (.-onload js/window) start)
```

会被转换成下面的JavaScript代码：

```
window.onload = hello_clojurescript.core.start;
```

由于服务器端的ClojureScript代码比较简单，这里不再进行介绍（如需了解更多细节，请参见本书配套代码）。

现在可以用`lein cljsbuild once`编译程序，并用`lein run`运行服务器。通过浏览器访问`http://localhost:3000`，就可以看到以下输出：

```
Hello from task 1
Hello from task 2
Hello from task 1
Hello from task 1
Hello from task 2
⋮
```

现在是否还觉得并发程序一定要依托于真正的多线程？

并发任务如果能一直独立运行那是极好的。但大多数界面需要和用户进行互动，这就要求代码能处理事件，下一节将学习如何处理事件。

处理事件

本节将通过一个简单的可以响应鼠标点击的动画，来演示ClojureScript是如何处理事件的。我们将创建一个页面，在页面上显示一些圆，这些圆会逐渐衰减成一个点，最终消失在用户鼠标点击的位置，如图6-4所示。

① https://developers.google.com/closure/library/

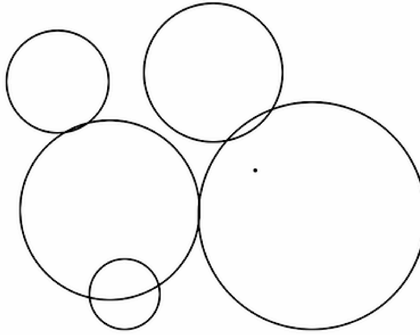


图6-4　衰减的圆

页面的代码非常简单，只使用一个<div>充满整个窗口：

CSP/Animation/resources/public/index.html

```
<html>
  <head>
    <title>Animation</title>
    <script src="/js/main.js" type="text/javascript"></script>
  </head>
  <body>
    <div id="canvas" width="100%" height="100%"></div>
  </body>
</html>
```

我们要借助Google Closure库的图形功能在页面上绘图，Google Closure库对不同浏览器上绘图操作的细节进行了抽象。`create-graphics`函数接受一个DOM元素，返回一个图像接口，通过这个图像接口可以在DOM元素上绘图：

CSP/Animation/src-cljs/animation/core.cljs

```
(defn create-graphics [elem]
  (doto (graphics/createGraphics "100%" "100%")
    (.render elem)))
```

`shrinking-circle`函数接受一个图形接口和一个位置，并创建一个go块，其绘制以输入的位置为中心的圆，并实现动画：

CSP/Animation/src-cljs/animation/core.cljs

```
Line 1 (def stroke (graphics/Stroke. 1 "#ff0000"))
     -
     - (defn shrinking-circle [graphics x y]
     -   (go
     5     (let [circle (.drawCircle graphics x y 100 stroke nil)]
     -       (loop [r 100]
     -         (<! (timeout 25))
     -         (.setRadius circle r r)
     -         (when (> r 0)
```

```
10          (recur (dec r))))
 -      (.dispose circle))))
```

这段代码首先使用Google Closure库的drawCircle函数（第5行）来绘制圆；然后进入循环，每25 ms调用一次setRadius函数（每秒40次）（第7行）；最后当半径衰减到0时，调用dispose函数来删除圆（第11行）。

我们还需要判断用户何时在页面上点击了鼠标。Google Closure库提供了listen函数，用于注册事件监听者。

类似于昨天学习的http/get函数，listen接受一个回调函数，在指定事件发生时会调用该回调函数。与昨天类似，我们传入一个将事件写入channel的回调函数，来将回调函数转换成core.async的形式：

CSP/Animation/src-cljs/animation/core.cljs

```
(defn get-events [elem event-type]
  (let [ch (chan)]
    (events/listen elem event-type
      #(put! ch %))
    ch))
```

还剩最后一点工作：

CSP/Animation/src-cljs/animation/core.cljs

```
(defn start []
  (let [canvas (dom/getElement "canvas")
        graphics (create-graphics canvas)
        clicks (get-events canvas "click")]
    (go (while true
          (let [click (<! clicks)
                x (.-offsetX click)
                y (.-offsetY click)]
            (shrinking-circle graphics x y))))))
(set! (.-onload js/window) start)
```

这段代码首先查找一个<div>作为画布，构造一个图形接口在画布上绘画，并获得鼠标点击事件构成的channel；然后进入循环，等待发生一次鼠标点击，获取这次点击的坐标offsetX和offsetY，并在这个坐标位置创建一个圆的动画。

这个例子看上去很简单（实际上也是这样），但它将JavaScript的基于回调的代码转换成了core.async的基于channel的代码，我们已经获得了巨大的成功——找到了“回调困境”的解决方案。

驯服回调

“回调困境”指的是由于JavaScript严重依赖回调方法而导致代码混乱的状况——回调函数调

用的回调函数再调用另一个回调函数，还需要暂存不同的状态来进行回调之间的通信。

下面将学习如何用本章介绍的异步编程模型来解决回调困境。

实现一个向导器①

向导器是常用的界面模式，指导用户一步一步地进行操作。今天最后的任务就是运用所学来创建一个不使用回调函数的向导器：

Wizard
localhost:3000/index.html
Step 2
Date of Birth: 03/10/1968
Homepage: www.paulbutcher.com
Next

图6-5　向导器

向导器由包含多个fieldset的表单构成：

CSP/Wizard/resources/public/index.html

```
<form id="wizard" action="/wizard" method="post">
  <fieldset class="step" id="step1">
    <legend>Step 1</legend>
    <label>First Name:</label><input type="text" name="firstname" />
    <label>Last Name:</label><input type="text" name="lastname" />
  </fieldset>

  <fieldset class="step" id="step2">
    <legend>Step 2</legend>
    <label>Date of Birth:</label><input type="date" name="dob" />
    <label>Homepage:</label><input type="url" name="url" />
  </fieldset>

  <fieldset class="step" id="step3">
    <legend>Step 3</legend>
    <label>Password:</label><input type="password" name="pass1" />
    <label>Confirm Password:</label><input type="password" name="pass2" />
  </fieldset>
  <input type="button" id="next" value="Next" />
</form>
```

① 有趣的是，原文标题是“We’re Off to See the Wizard”，这是绿野仙踪的一首插曲。“wizard”既有“巫师”的意思，也有“向导器”的意思。——译者注

每个<fieldset>代表向导器的一个步骤。先将所有的fieldset隐藏起来：

CSP/Wizard/resources/public/styles.css

```
label { display:block; width:8em; clear:left; float:left;
      text-align:right; margin-right: 3pt; }
input { display:block; }
➤ .step { display:none; }
```

之后的程序会使用下面这些辅助函数，显示或隐藏相关的fieldset：

CSP/Wizard/src-cljs/wizard/core.cljs

```
(defn show [elem]
  (set! (.. elem -style -display) "block"))

(defn hide [elem]
  (set! (.. elem -style -display) "none"))

(defn set-value [elem value]
  (set! (.-value elem) value))
```

这段代码使用了另一种形式的dot special form，用于访问对象深层的属性，比如：

```
(set! (.. elem -style -display) "block")
```

会被转换成下面的JavaScript代码：

```
elem.style.display = "block";
```

下面的代码实现了向导器的控制流程：

CSP/Wizard/src-cljs/wizard/core.cljs

```
Line 1 (defn start []
     -  (go
     -    (let [wizard (dom/getElement "wizard")
     -          step1 (dom/getElement "step1")
     5          step2 (dom/getElement "step2")
     -          step3 (dom/getElement "step3")
     -          next-button (dom/getElement "next")
     -          next-clicks (get-events next-button "click")]
     -      (show step1)
    10      (<! next-clicks)
     -      (hide step1)
     -      (show step2)
     -      (<! next-clicks)
     -      (set-value next-button "Finish")
    15      (hide step2)
     -      (show step3)
     -      (<! next-clicks)
     -      (.submit wizard))))
     -
    20 (set! (.-onload js/window) start)
```

这段代码首先获取了表单中每个需要操作的元素的引用，并用之前写的`get-events`函数获取Next按钮的点击事件的channel（第8行）；然后显示与向导第一步相关的表单元素，并等待用户点击Next按钮（第10行）；当用户点击时，隐藏与向导第一步相关的表单元素，显示与向导第二步相关的表单元素，并等待用户再次点击Next按钮；以此类推，直到所有步骤都完成，最后提交表单（第18行）。

这段代码最大的特点是它没什么特别之处——向导器的流程是串行的，这段代码看上去也是串行的。当然这是由于`go`宏的作用，实际上这段代码并不是串行的——我们创建了一个状态机，它或者处于运行状态，或者在等待状态转换信号时处于暂停状态。绝大多数时候我们不需要关心细节，只需要将其视作串行代码即可。

第三天总结

我们完成了第三天的学习，同时也结束了对`core.async`版本的CSP模型的讨论。

第三天我们学到了什么

ClojureScript是Clojure的一个子集，其代码可以编译成JavaScript，这样客户端编程就可以借助`core.async`的力量。这不仅可以在单线程JavaScript环境上运行协作式多任务，还能解决“回调困境”。

第三天自习

查找

- ClojureScript中，`core.async`支持暂停函数，比如`<!`和`>!`，但并不支持阻塞函数`<!!`和`>!!`。这是为什么？
- 阅读`take!`的文档，通过这个函数，如何将基于channel的API转换为基于回调函数的API？这适用于什么场景？（提示：这个问题与上一个问题相关。）

实践

- 用`core.async`编写一个简单的浏览器游戏，比如贪吃蛇、乒乓球或打砖块。
- 用JavaScript重写向导器的例子。其与ClojureScript版本相比有什么区别？

6.5　复习

表面上看，actor模型和CSP模型非常相似——它们均由独立的并发的执行单元构成，这些执行单元之间都使用消息来进行通信。但如本章所述，由于侧重点不同，这两种模型可谓是大相径庭。

优点

与actor模型相比，CSP模型的最大的优点是灵活性。使用actor模型时，负责通信的媒介与执行单元是紧耦合的——每个actor都有一个信箱。而使用CSP模型时，channel是第一类对象，可以被独立地创建、写入数据、读出数据，也可以在不同的执行单元中传递。

Clojure语言的创始人Rich Hickey解释了他在CSP模型与actor模型中选择前者的原因[①]：

> 我个人对actor模型并不感兴趣。在actor模型中，生产者和消费者还是耦合在一起的。诚然，我们可以用actor模型实现消息通信用的队列（人们确实也经常这样做），但actor模型本身就已经使用了队列，用它来实现基础的消息通信用的队列未免显得画蛇添足。[②]

从更务实的角度来说，现在的CSP模型的实现，比如`core.async`库，使用了控制反转技术，不仅提高了异步程序的效率，还为原本使用回调函数来解决的应用领域提供了一种显著改进的编程模型。本章中我们学习了其中的两种模型：异步IO编程和异步UI编程，然而还有很多其他模型。

缺点

如果将本章与上一章相比，有两个主题没有被提及——分布式和容错性。基于CSP模型的编程语言虽然也可以支持分布式和容错性，但与基于actor模型的编程语言不同，这两个主题并没有得到足够的重视和支持——也没有基于CSP模型实现的OTP。

与使用线程与锁模型和actor模型一样，CSP模型也容易受到死锁影响，且没有提供直接的并行支持。使用CSP模型时，并行需要建立在并发的基础上，这也就引入了不确定性。

其他语言

与actor模型类似，CSP模型由Tony Hoare于1970年代提出。近年来这两个模型一直在互相借鉴，共同进化。

20世纪80年代，编程语言occam[③]使用CSP模型为基石。然而在当下，Go语言是使用CSP模型的最流行的语言。

`core.async`和Go语言都使用控制反转来实现异步任务，这一技术也被其他语言广泛使用，包括F#[④]、C#[⑤]、Nemerle[⑥]和Scala[⑦]。

---

① http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html

② Rich Hickey在文章中主要阐述了设计`core.async`的目的，其中一个目的是提供消息通信用的队列。这段引文正是从提供消息通信用的队列这一角度来阐述选择CSP模型的原因。——译者注

③ http://en.wikipedia.org/wiki/Occam_programming_language

④ http://blogs.msdn.com/b/dsyme/archive/2007/10/11/introducing-f-asynchronous-workflows.aspx

⑤ http://msdn.microsoft.com/en-us/library/hh191443.aspx

⑥ https://github.com/rsdn/nemerle/wiki/Computation-Expression-macro#async

⑦ http://docs.scala-lang.org/sips/pending/async.html

结语

CSP模型和actor模型开发社区的侧重点不同，并各自发展，从而形成了两者之间的诸多差异。actor模型的开发社区侧重于容错性和分布式，而CSP模型的开发社区侧重于效率和代码表达的流畅性。如何在这两种模型之间进行选择，很大程度上取决于你关注于哪个方面。

CSP模型是本书介绍的最后一种通用的编程模型。下一章我们将学习第一个专用的编程模型。

第 7 章

数据并行

7

数据并行就像是八车道的高速公路。虽然每辆车的速度相对平缓，但由于多辆车可以同时行进，所以通过某一点的车流量还是很大的。

到目前为止，我们讨论的每一项技术都可以用于解决多种编程问题。相比之下，数据并行只适用于很窄的范围。顾名思义，数据并行是并行编程技术，而不是并发编程技术（并发和并行是相关的，但又有所区别，参见1.1节）。

7.1 隐藏在笔记本电脑中的超级计算机

本章我们将学习如何利用隐藏在笔记本电脑中的超级计算机——图形处理单元（GPU）。现代GPU是一个强力的数据并行处理器，其用于数学计算时性能超过了CPU，这种做法称为基于图形处理器的通用计算（General-Purpose computing on the GPU），或GPGPU编程。

过去数年间，有许多技术致力于将不同GPU的实现细节抽象出来。本书将使用开放计算语言（OpenCL）[①]来编写GPGPU代码。

第一天，我们将学习编写OpenCL内核的基础知识，以及用于编译和运行内核的主机程序。第二天，学习如何将内核映射到硬件上。第三天，学习OpenCL如何与开放图形库（OpenGL）[②]写就的图形代码进行交互。

7.2 第一天：GPGPU 编程

今天我们将学习如何编写一个简单的将两个数组并行相乘的GPGPU程序，并测试这个程序的性能，来看看GPU与CPU相比性能会有多大提升。不过我们要先花点时间来探究一下为什么GPU在进行数学运算时比较强力。

① http://www.khronos.org/opencl/

② http://www.opengl.org

图形处理与数据并行

计算机图形学主要研究如何处理数据、如何处理大量数据以及如何快速处理大量数据。3D游戏的一个场景是由无数个小三角形构成的，每一个三角形都需要根据与视点相关的透视关系计算出其在屏幕上的位置，并进行裁剪、处理光照、修饰纹理等，这些操作每秒钟都要进行25次以上。

虽然需要处理的数据量是巨大的，但其有一个非常好的特性：施加在数据上的操作都是相对简单的向量操作或矩阵操作。因此这种场景非常适合于*数据并行*——多个计算资源会在不同的数据上并行地施加同样的操作。

现代GPU是异常复杂但十分强力的并行处理器，其1秒钟可以处理几十亿个三角形。虽然设计GPU的主要目的是为了满足图形计算的需要，但是GPU也可用于更广的领域。

数据并行可以通过多种方法来实现。我们要学习其中的两种：流水线和多ALU。

流水线

虽然看上去将两数相乘是一个原子操作，但如果从芯片上的门电路的角度来看，这个操作实际上是分几步完成的。这些步骤通常被排列成流水线型，如图7-1所示。

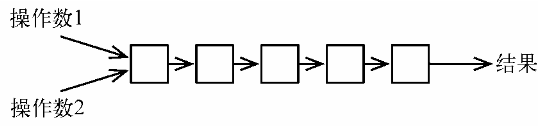


图7-1　两个数乘法的操作流水线

图7-1是一个有五个步骤的流水线，如果每一步骤需要1个时钟周期来完成，那将一组数（两个数）相乘就需要5个时钟周期。但如果有多组数要相乘，就可以通过让流水线饱和来获得更好的性能（假设内存子系统足够快，可以及时提供足够多的数），如图7-2所示。

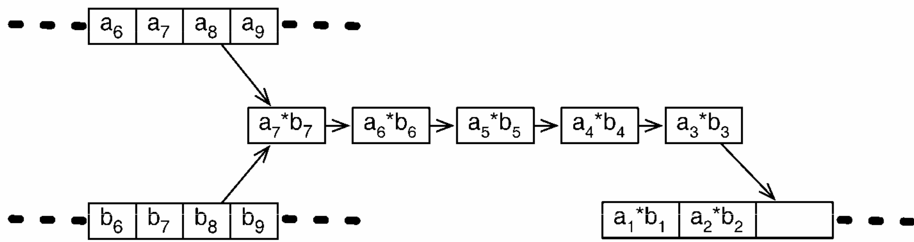


图7-2　多组数乘法的饱和的操作流水线

如果需要将1000组数相乘，每组数需要5个时钟周期，看上去总共需要5000个时钟周期，而如图7-2所示，实际上只需要略多于1000个时钟周期即可完成。

多ALU

CPU中负责进行乘法之类运算的组件称为算术逻辑单元（ALU），如图7-3所示。

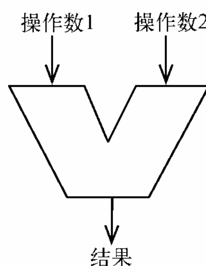


图7-3 ALU

只要搭配足够宽的内存总线，多个ALU就可以同时获取多个操作数，这样施加在大量数据上的运算就可以并行了，如图7-4所示。

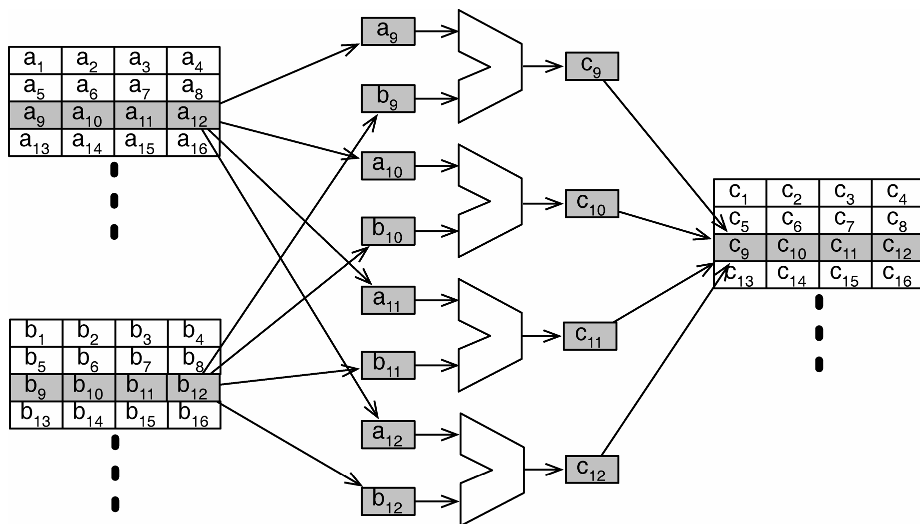


图7-4 利用多个ALU对大量数据进行并行操作

GPU的内存总线通常有256位或更宽，也就是说一次可以获取8个或更多个32位的浮点数。

混乱的局面

为了获得更好的性能，现实中的GPU会综合使用流水线、多ALU以及许多本书尚未提及的技术。这就增加了进一步理解GPU的难度。更遗憾的是，不同的（即使是同一厂商生产的）GPU之间的共性是很少的。如果我们必须针对某个架构开发代码，GPGPU编程不是最佳选择。

OpenCL定义了一种类C语言，可以针对多种架构抽象地进行编程。不同的GPU厂商会提供各自的编译器和驱动程序，使代码可以被编译并在相应的硬件上运行。

第一个 OpenCL 程序

如果要利用OpenCL对数组相乘进行并行化，就需要先将工作划分成多个可以并行执行的工作项（work-item）。

工作项

在编写并行代码时，需要留意每个并行任务的颗粒度。通常，如果每个任务只进行非常少的工作，代码运行的效率会很低，其原因是线程创建和线程通信的代价会变得很高。

相比之下，OpenCL的工作项通常会很小。如果要将两个有1024个元素的数组相乘，就可以创建1024个工作项。如图7-5所示。

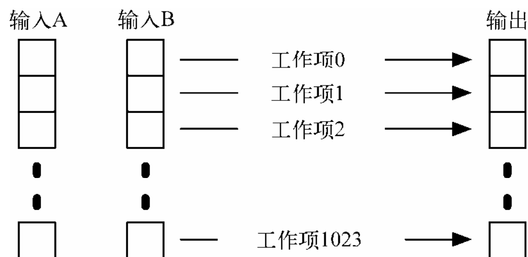


图7-5 数组相乘的工作项

我们的任务就是将问题拆分成尽可能小的工作项。OpenCL的编译器和运行时会根据硬件条件对工作项进行最优调度，保证硬件被尽可能高效地利用。

OpenCL调优

如你所料，现实世界往往不是这么简单。对OpenCL程序进行调优，通常需要仔细考量可用的资源，并给编译器和运行时一些提示，帮助它们更好地调度工作项。有时为了性能甚至需要限制并行度。

不过，过早地进行调优是编程之大忌。大部分情况下，只需要让并行最大化、让工作项最小化，仅在必要的时候才会考虑进行优化。

内核

OpenCL的内核主要用于说明每个工作项是如何工作的。下面是数组相乘程序的内核：

DataParallelism/MultiplyArrays/multiply_arrays.cl

```
__kernel void multiply_arrays(__global const float* inputA,
                              __global const float* inputB,
                              __global float* output) {

  int i = get_global_id(0);
  output[i] = inputA[i] * inputB[i];
}
```

这个内核接受两个输入数组指针inputA和inputB，和一个输出数组指针output。它调用get_global_id()来确定当前正在处理哪个工作项，并将inputA和inputB的对应元素相乘的结果写入output的对应元素。

要创建一个完整的程序，就需要将这个内核嵌入到主机程序中，步骤如下：

(1) 创建上下文，内核和命令队列都将运行在这个上下文中；
(2) 编译内核；
(3) 创建输入数据的缓存区和输出数据的缓存区；
(4) 向命令队列中输入一个命令，让每一个工作项都运行一次内核程序；
(5) 获取结果。

OpenCL官方标准定义了C绑定（binding）和C++绑定。然而，大部分主流语言都提供了非官方的绑定,所以我们可以用自己喜欢的语言来编写主机程序。由于OpenCL官方标准使用了C语言，而且C语言能最好地描述底层的运行原理，因此我们一开始先选择C语言，第三天再学习用Java编写主机程序。

接下来的几节将会完成一个完整的OpenCL主机程序。为了在代码中更清楚地表达出底层的数据结构，我们将写一些奔放的不带错误处理的代码——不过别担心，稍后会对这些代码进行修正。你还会发现在函数参数中使用了多个NULL——同样别担心，在理解了整体结构后，我们会回头仔细学习这些API。

创建上下文

OpenCL上下文表示了OpenCL内核运行的环境。要创建上下文，需要识别所使用的平台，以及平台上将运行内核的设备（稍后会详细讨论平台和设备）：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
```

这段代码定义了一个简单的上下文，其仅包含一个GPU。首先调用clGetPlatformIDs()来识别平台，然后将CL_DEVICE_TYPE_GPU传给clGetDeviceIDs()来获取GPU的ID，最后将此ID

传给clCreateContext()来创建上下文。

创建命令队列

已经创建了一个上下文，现在可以用它创建命令队列了：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);
```

clCreateCommandQueue()接受一个上下文和一个设备ID，并返回一个命令队列，命令将通过这个队列发送给该设备。

编译内核

接下来要将内核编译成可以在设备上运行的代码：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
char* source = read_source("multiply_arrays.cl");
cl_program program = clCreateProgramWithSource(context, 1,
  (const char**)&source, NULL, NULL);
free(source);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "multiply_arrays", NULL);
```

这段代码首先将multiply_arrays.cl中的内核源码读到一个字符串中（在本书配套代码中可以看到read_source()的源码），然后将该字符串传给clCreateProgramWithSource()，之后使用clBuildProgram()进行构建，最后使用clCreateKernel()创建一个内核。

创建缓存区

内核使用的是缓存区中的数据，我们先来创建缓存区：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
#define NUM_ELEMENTS 1024

cl_float a[NUM_ELEMENTS], b[NUM_ELEMENTS];
random_fill(a, NUM_ELEMENTS);
random_fill(b, NUM_ELEMENTS);
cl_mem inputA = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(cl_float) * NUM_ELEMENTS, a, NULL);
cl_mem inputB = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  sizeof(cl_float) * NUM_ELEMENTS, b, NULL);
cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
  sizeof(cl_float) * NUM_ELEMENTS, NULL, NULL);
```

这段代码创建了两个数组a和b，并调用random_fill()向两个数组中灌入随机数：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
void random_fill(cl_float array[], size_t size) {
  for (int i = 0; i < size; ++i)
```

```
    array[i] = (cl_float)rand() / RAND_MAX;
}
```

从内核的角度来看，两个输入缓存区inputA和inputB都是只读的（CL_MEM_READ_ONLY），并从对应的数组中复制数据作为初始值（CL_MEM_COPY_HOST_PTR）。输出缓存区output是只写的（CL_MEM_WRITE_ONLY）。

执行工作项

终于可以执行工作项来进行数组相乘了：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &inputB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);

size_t work_units = NUM_ELEMENTS;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &work_units, NULL, 0, NULL, NULL);
```

这段代码首先调用clSetKernelArg()来设置内核参数，然后调用clEnqueueNDRangeKernel()，将*N*维空间（NDRange）的工作项传给命令队列。本例中*N*=1（clEnqueueNDRangeKernel()的第三个参数——稍后会看到*N*>1的例子），工作项的个数是1024。

获取结果

当内核运行结束，就可以获取结果了：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
cl_float results[NUM_ELEMENTS];
clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(cl_float) * NUM_ELEMENTS,
  results, 0, NULL, NULL);
```

这段代码创建了result数组，并调用clEnqueueReadBuffer()将output缓存区的内容复制到该数组中。

清理

主机程序最后的工作是清理现场：

DataParallelism/MultiplyArrays/multiply_arrays.c

```
clReleaseMemObject(inputA);
clReleaseMemObject(inputB);
clReleaseMemObject(output);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

性能分析

我们已经得到了一个内核，现在该来分析一下其性能了。这次需要使用OpenCL性能分析API：

DataParallelism/MultiplyArraysProfiled/multiply_arrays.c

```
Line 1 cl_event timing_event;
     - size_t work_units = NUM_ELEMENTS;
     - clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &work_units,
     -   NULL, 0, NULL, &timing_event);
     5
     - cl_float results[NUM_ELEMENTS];
     - clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(cl_float) * NUM_ELEMENTS,
     -   results, 0, NULL, NULL);
     - cl_ulong starttime;
    10 clGetEventProfilingInfo(timing_event, CL_PROFILING_COMMAND_START,
     -   sizeof(cl_ulong), &starttime, NULL);
     - cl_ulong endtime;
     - clGetEventProfilingInfo(timing_event, CL_PROFILING_COMMAND_END,
     -   sizeof(cl_ulong), &endtime, NULL);
    15 printf("Elapsed (GPU): %lu ns\n\n", (unsigned long)(endtime - starttime));
     - clReleaseEvent(timing_event);
```

这段代码将事件`timing_event`传给`clEnqueueNDRangeKernel()`（第3行）。当一个命令完成时，就可以调用`clGetEventProfilingInfo()`来查看记录在这个事件中的时间信息（第10行和第13行）。

如果将`NUM_ELEMENTS`设置为100 000，我的MacBook Pro的GPU会花费43 000纳秒。而如果使用简单循环在CPU中进行同样的操作：

DataParallelism/MultiplyArraysProfiled/multiply_arrays.c

```
for (int i = 0; i < NUM_ELEMENTS; ++i)
  results[i] = a[i] * b[i];
```

将同样的100 000个元素相乘，需要花费近400 000纳秒，可见执行这个任务时GPU比CPU快9倍。

注意事项

对数组相乘进行性能分析可能会造成一些误区。在执行命令之前，程序将输入数据复制到`inputA`和`inputB`缓存区中；在命令完成后，程序又从`output`缓存区中将结果复制出来。

对于数组相乘这种简单任务来说，这些数据复制的成本相对较高，以至于会影响到整个性能分析的结果。实际使用中，OpenCL应用通常会对操作数进行更复杂的操作，或者是对已经在GPU上的数进行操作。

为简单起见，本例并没有涉及OpenCL API的细节。现在是时候讨论它们了。

多返回值

很多OpenCL函数都会返回多个值。如果一个平台支持多个设备，clGetDeviceIDs()函数就会返回多个设备。其函数原型如下：

```
cl_int clGetDeviceIDs(cl_platform_id platform,
                      cl_device_type device_type,
                      cl_uint        num_entries,
                      cl_device_id*  devices,
                      cl_uint*       num_devices);
```

参数devices是一个长度为num_entries的数组的指针，参数num_devices是一个整数的指针。调用clGetDeviceIDs()的一种方法是使用定长数组：

```
cl_device_id devices[8];
cl_uint num_devices;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 8, devices, &num_devices);
```

clGetDeviceIDs()返回时，num_devices已经被赋值为可用设备的个数，devices的前num_devices个元素也会被设置好。

这看似不错，但如果可用设备超过8个呢？当然可以使用一个“大”数组，但经验告诉我们无论设置多大的数组，总有一天会超过这个极值。幸运的是，所有返回数组的OpenCL函数都提供了一种方式，可以返回任意大的数组——两次调用函数：

```
cl_uint num_devices;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);

cl_device_id* devices = (cl_device_id*)malloc(sizeof(cl_device_id) * num_devices);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);
```

第一次调用clGetDeviceIDs()时，用NULL作为参数devices。其返回时，num_devices已经被设置成可用设备的个数。接下来只需要创建一个合适长度的数组，并第二次调用clGetDeviceIDs()。

错误处理

7

OpenCL函数通过错误码来报错。CL_SUCCESS表示函数运行成功；其他的代码表示函数运行失败。调用clGetDeviceIDs()并进行错误处理的代码如下：

```
cl_int status;

cl_uint num_devices;
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);
if (status != CL_SUCCESS) {
  fprintf(stderr, "Error: unable to determine num_devices (%d)\n", status);
  exit(1);
}
```

```
cl_device_id* devices = (cl_device_id*)malloc(sizeof(cl_device_id) * num_devices);
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);
if (status != CL_SUCCESS) {
  fprintf(stderr, "Error: unable to retrieve devices (%d)\n", status);
  exit(1);
}
```

显而易见，OpenCL代码会借助辅助函数或辅助宏来消除重复代码，类似于：

DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c

```
#define CHECK_STATUS(s) do { \
    cl_int ss = (s); \
    if (ss != CL_SUCCESS) { \
      fprintf(stderr, "Error %d at line %d\n", ss, __LINE__); \
      exit(1); \
    } \
} while (0)
```

使用这个辅助宏：

DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c

```
CHECK_STATUS(clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA));
```

有些函数不返回错误码，而是接受`error_ret`参数。比如`clCreateContext()`的函数原型：

```
cl_context clCreateContext(const cl_context_properties* properties,
                           cl_uint num_devices,
                           const cl_device_id* devices,
                           void (CL_CALLBACK* pfn_notify)(const char* errinfo,
                                                          const void* private_info,
                                                          size_t cb,
                                                          void* user_data),
                           void* user_data,
                           cl_int* errcode_ret);
```

对这个函数的错误处理如下：

DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c

```
cl_int status;
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
CHECK_STATUS(status);
```

OpenCL代码中还有其他一些常用的错误处理方式——你应该选择一种最适合自己的方式。

第一天总结

我们结束了第一天的学习。第二天将深入了解OpenCL平台、执行和内存模型。

第一天我们学到了什么

利用OpenCL可以挖掘出GPU的数据并行处理的能力，并进行通用编程，从而获得很大的性能提升。

- 通过将任务切分成工作项，OpenCL可以将任务并行化。
- 通过编写内核，指定了单个工作项是如何工作的。
- 要执行内核，主机程序必须遵守以下步骤：

  (1) 创建上下文，内核和命令队列都将运行在这个上下文中；
  (2) 编译内核；
  (3) 创建输入数据的缓存区和输出数据的缓存区；
  (4) 向命令队列中输入一个命令，让每一个工作项都运行一次内核程序；
  (5) 获取结果。

第一天自习

查找

- 阅读OpenCL标准（The OpenCL specification）。
- 阅读OpenCL API参考卡片（The OpenCL API reference card）。
- 定义OpenCL内核的是类C语言。它和C语言有什么不同？

实践

- 修改数组相乘的内核，使其能处理不同类型的数组，并分析其性能。处理不同数据类型时性能是否有差异？数据类型的长度（字节数）是否会影响性能？是否会影响与CPU的性能比较结果？
- 之前我们将`CL_MEM_COPY_HOST_PTR`传给`clCreateBuffer()`来创建并初始化缓存区。修改主机代码，使用`CL_MEM_USE_HOST_PTR`或`CL_MEM_ALLOC_HOST_PTR`（除了修改参数，你还需要修改一些代码），并分析性能。如何权衡不同缓存分配策略的使用？
- 修改主机代码，用`clEnqueueMapBuffer()`替换`clCreateBuffer()`，并分析性能。其适用于何种场景？不适用于何种场景？
- 在标准C的基础上，OpenCL还提供了大量的数据类型——特别是`float4`和`ulong3`这类向量类型。修改内核代码，进行两组向量的相乘。这种向量类型在主机端应该如何表示？

7.3　第二天：多维空间与工作组

昨天我们学习了如何用`clEnqueueNDRangeKernel()`执行多个工作项，来进行一维数组的运算。今天将扩展到多维数组的运算，并利用OpenCL的工作组来解决更大规模的问题。

多维工作项空间

当主机程序调用`clEnqueueNDRangeKernel()`执行内核时，就定义了一个索引空间，其中的每个点都有全局唯一的ID，并代表一个工作项。

内核通过调用`get_global_id()`来查找当前正在处理的工作项的全局ID。昨天的例子中索引空间是一维的，因此内核只需要调用`get_global_id()`一次。今天我们会创建一个进行二维数组乘法的内核，其需要调用`get_global_id()`两次。

矩阵乘法

让我们穿越回大学时代的线性代数课堂，重温一下如何进行矩阵乘法。

矩阵是一个二维数组。如果将一个$w \times n$的矩阵[①]与一个$m \times w$的矩阵相乘（注意第一个矩阵的列数一定要等于第二个矩阵的行数），就会得到一个$m \times n$的矩阵。例如，将一个2×4的矩阵与一个3×2的矩阵相乘会得到一个3×4的矩阵。

为了计算输出矩阵上位置为(i, j)的元素[②]的值，需要将第一个矩阵的第j行的每一个元素与第二个矩阵第i列的对应元素相乘，并将所有乘积相加。

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{pmatrix}$$

下面是进行矩阵乘法的串行执行代码：

```
#define WIDTH_OUTPUT WIDTH_B
#define HEIGHT_OUTPUT HEIGHT_A

float a[HEIGHT_A][WIDTH_A] = «initialize a»;
float b[HEIGHT_B][WIDTH_B] = «initialize b»;
float r[HEIGHT_OUTPUT][WIDTH_OUTPUT];

for (int j = 0; j < HEIGHT_OUTPUT; ++j) {
  for (int i = 0; i < WIDTH_OUTPUT; ++i) {
    float sum = 0.0;
    for (int k = 0; k < WIDTH_A; ++k) {
      sum += a[j][k] * b[k][i];
    }
    r[j][i] = sum;
  }
}
```

① 作者本章中的$w \times n$表示的是一个n行w列的矩阵，而通常我们会用$w \times n$表示一个w行n列的矩阵。我们将沿用作者的标记方法，请读者注意此差别。——译者注

② 作者本章中的(i, j)表示的是矩阵的第j行第i列的元素，而通常我们会用(i, j)表示矩阵的第i行第j列的元素。我们将沿用作者的标记方法，请读者注意此差别。——译者注

显而易见，随着矩阵规模增加，矩阵相乘的计算量会猛增，大矩阵的乘法确实是非常消耗CPU的。

并行矩阵乘法

下面是用于矩阵乘法的内核代码：

DataParallelism/MatrixMultiplication/matrix_multiplication.cl

```
Line 1 __kernel void matrix_multiplication(uint widthA,
     -                                    __global const float* inputA,
     -                                    __global const float* inputB,
     -                                    __global float* output) {
     5
     -   int i = get_global_id(0);
     -   int j = get_global_id(1);
     -
     -   int outputWidth = get_global_size(0);
    10   int outputHeight = get_global_size(1);
     -   int widthB = outputWidth;
     -
     -   float total = 0.0;
     -   for (int k = 0; k < widthA; ++k) {
    15     total += inputA[j * widthA + k] * inputB[k * widthB + i];
     -    }
     -    output[j * outputWidth + i] = total;
     - }
```

这个内核是在一个二维索引空间中运行，索引空间中的每个位置都代表输出矩阵中的一个元素。内核通过两次调用`get_global_id()`函数来获取当前的位置（第6、7行）。

调用`get_global_size()`可以获得索引空间的大小，内核利用这个特性可以得到输出矩阵的大小（第9、10行）。同时也得到了`widthB`，因为它与`outputWidth`相等。不过还是要依靠参数传入`widthA`。

第14行的循环是串行版本代码中的内循环——唯一的区别是，由于OpenCL的缓存区不能是多维的，因此没法写出下面这样的代码：

```
output[j][i] = total;
```

但可以用一个简单的计算来确定数组偏移：

```
output[j * outputWidth + i] = total;
```

这个内核对应的主机程序与昨天的很相似，唯一的区别是调用`clEnqueueNDRangeKernel()`时的参数：

DataParallelism/MatrixMultiplication/matrix_multiplication.c

```
size_t work_units[] = {WIDTH_OUTPUT, HEIGHT_OUTPUT};
CHECK_STATUS(clEnqueueNDRangeKernel(queue, kernel, 2, NULL, work_units,
  NULL, 0, NULL, NULL));
```

要创建二维的索引空间，需要设置`work_dim`（第3个参数）的值为2，并将`global_work_size`（第5个参数）设置为一个二元数组，该二元数组描述了每一维度的大小。

比起昨天的性能分析结果，这个内核有更大的性能提升。在我的Macbook Pro上将200 × 400的矩阵与300 × 200的矩阵相乘，CPU花费了66 ms，而GPU花费了3 ms，性能提升了20多倍。

由于这个内核花费了很大的工作量在处理数据上，因此即使将复制数据的成本考虑进来，仍能获得很大的性能提升。在我的Macbook Pro上，数据复制需要花费2 ms，那GPU花费的总时间为5 ms，仍有13倍的性能提升。

到目前为止，本书一直在使用一个假想的兼容OpenCL标准的GPU。这显然不太现实，因此需要了解如何确定一个主机兼容哪种OpenCL平台和设备。

查询设备信息

OpenCL提供了多种用于查询平台参数、设备和其他API对象的函数。例如下面这个函数，其通过`clGetDeviceInfo()`来查询一个设备参数，并以字符串形式输出：

DataParallelism/FindDevices/find_devices.c

```
void print_device_param_string(cl_device_id device,
                               cl_device_info param_id,
                               const char* param_name) {
  char value[1024];
  CHECK_STATUS(clGetDeviceInfo(device, param_id, sizeof(value), value, NULL));
  printf("%s: %s\n", param_name, value);
}
```

不同参数的类型是不一样的（字符串、整数、`size_t`数组等）。通过一系列类似的函数，可以查询一个指定设备的参数：

DataParallelism/FindDevices/find_devices.c

```
void print_device_info(cl_device_id device) {
  print_device_param_string(device, CL_DEVICE_NAME, "Name");
  print_device_param_string(device, CL_DEVICE_VENDOR, "Vendor");
  print_device_param_uint(device, CL_DEVICE_MAX_COMPUTE_UNITS, "Compute Units");
  print_device_param_ulong(device, CL_DEVICE_GLOBAL_MEM_SIZE, "Global Memory");
  print_device_param_ulong(device, CL_DEVICE_LOCAL_MEM_SIZE, "Local Memory");
  print_device_param_sizet(device, CL_DEVICE_MAX_WORK_GROUP_SIZE, "Workgroup size");
}
```

本书配套代码中有一个`find_devices`程序就是使用这样的代码来查询可用的平台和设备的。在我的Macbook Pro上运行这段程序会得到以下输出：

```
Found 1 OpenCL platform(s)

Platform 0
Name: Apple
```

```
Vendor: Apple

Found 2 device(s)

Device 0
Name: Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
Vendor: Intel
Compute Units: 8
Global Memory: 17179869184
Local Memory: 32768
Workgroup size: 1024
Device 1
Name: GeForce GT 650M
Vendor: NVIDIA
Compute Units: 2
Global Memory: 1073741824
Local Memory: 49152
Workgroup size: 1024
```

其中有一个可用的平台，就是默认的Apple OpenCL实现。这个平台有两个设备：CPU和GPU。

我们还发现一些有趣的事情：

- OpenCL不只适用于GPU（还适用于CPU，以及专用OpenCL加速器）；
- 我的Macbook Pro的GPU有两个计算单元（compute unit）（稍后会学习什么是计算单元）；
- 这个GPU的全局内存（global memory）有1 GiB；
- 每个计算单元的局部内存（local memory）有48 KiB，可支持的工作组的最大规模为1024。

下面几节将介绍OpenCL的平台模型和内存模型，以及它们对代码的影响。



小乔爱问：

为什么OpenCL会适用于CPU?

OpenCL 适用于 CPU，这是很多人没有想到的。事实上现代 CPU 支持数据并行指令已经很长时间了。例如 Intel 处理器就支持流式 SIMD 扩展指令集（Streaming SIMD Extensions，SSE）和高级矢量扩展指令集（Advanced Vector Extensions，AVX）。OpenCL 可以高效地使用这些扩展指令集和多核 CPU。

平台模型

OpenCL平台由连接到一个或多个设备的主机构成。每个设备都有一个或多个计算单元，每个计算单元提供很多处理元件（processing element），如图7-6所示。

7

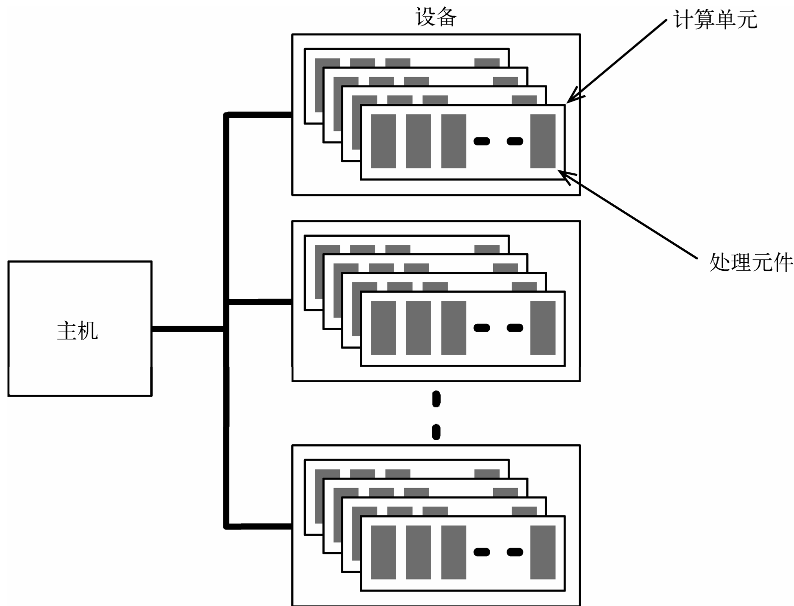


图7-6　OpenCL平台模型

工作项是在处理元件中执行的。在同一个计算单元中执行的工作项的集合称为工作组。一个工作组中的工作项共享使用局部内存。下面将介绍OpenCL的内存模型。

内存模型

工作项执行内核程序时，会访问四种不同的内存区域。

全局内存（Global memory）：同一个设备上执行的所有工作项都可以使用的内存。

常量内存（Constant memory）：全局内存的一部分，在执行内核时保持不变。

局部内存（Local memory）：工作组私有的内存，可用于工作组中不同工作项之间的通信（稍后会举例说明）。

私有内存（Private memory）：工作项私有的内存。

前几章介绍过集合的化简操作，化简操作可以用于解决很多问题。下一节将介绍如何实现数据并行版的化简操作。



小乔爱问：

OpenCL设备真的是这样工作的吗？

OpenCL 的平台模型和内存模型并不限制底层硬件的工作方式。它们只是底层硬件的一种抽象——不同的 OpenCL 设备有多种不同的物理架构。

例如，某种 OpenCL 设备的局部内存是计算单元私有的，而另一种设备的局部内存却是映射到全局内存的一个区域；某种设备是有独立的全局内存的，而另一种设备则是直接访问主机内存的。

这些架构上的差异在优化 OpenCL 代码时意义重大，不过这超出了本章的范围。

使用数据并行进行化简操作

本节将创建一个查找集合最小元素的内核，其使用`min()`函数对集合进行化简。

先用串行编程来实现：

DataParallelism/FindMinimumOneWorkGroup/find_minimum.c

```
cl_float acc = FLT_MAX;
for (int i = 0; i < NUM_VALUES; ++i)
  acc = fmin(acc, values[i]);
```

分作两步将其并行化——第一步使用一个工作组，第二步使用多个工作组。

使用一个工作组进行化简操作

为叙述方便（稍后会解释其原因），进行以下假设：其一，要化简的数组的元素个数是2的乘方；其二，要化简的数组的元素个数不能太多，以便一个工作组就可以处理。在这种假设下，实现化简操作的内核为：

DataParallelism/FindMinimumOneWorkGroup/find_minimum.cl

```
Line 1 __kernel void find_minimum(__global const Line 1 float* values,
     -                            __global float* result,
     -                            __local float* scratch) {
     -   int i = get_global_id(0);
     5   int n = get_global_size(0);
     -   scratch[i] = values[i];
     -   barrier(CLK_LOCAL_MEM_FENCE);
     -   for (int j = n / 2; j > 0; j /= 2) {
     -     if (i < j)
    10       scratch[i] = min(scratch[i], scratch[i + j]);
     -     barrier(CLK_LOCAL_MEM_FENCE);
     -   }
```

```
-    if (i == 0)
-      *result = scratch[0];
15 }
```

上面的算法分为三个阶段：

(1) 从全局内存向局部内存（scratch）复制数组（第6行）；(2) 进行化简操作（第8~12行）；(3) 将结果复制到全局内存中（第14行）。

化简操作是按照一个树形顺序进行的，这棵树非常像我们在介绍Clojure的reducer时见过的树（参见3.3节），如图7-7所示。

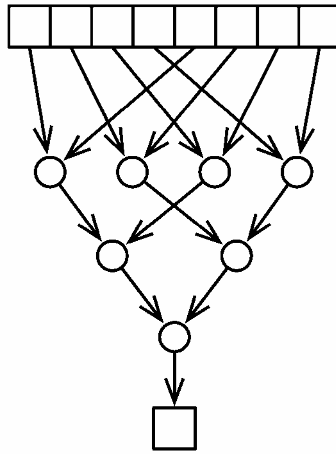


图7-7　化简操作的树形顺序

每循环一次，有一半的工作项将失去活性——原因是只有i<j的工作项才会进行操作（这就是为什么要假设数组的元素个数是2的乘方——这样就能一直对数组进行二分而不用担心边界情况）。当只剩下一个活动的工作项时退出循环。每个活动的工作项在每个循环中都会进行一次min()操作，比较当前元素与另一半数组中的对应元素，并取较小者。当退出循环时，scratch数组的第一项就是化简操作的结果，工作组的第一个工作项负责将这个结果复制到result中。

这个内核另一个有趣的地方是其使用了同步屏障（barrier）（第7行、第11行）来同步对局部内存的访问。

同步屏障

同步屏障（barrier）是一种同步手段，用来协调多个工作项对局部内存的使用。如果工作组中一个工作项执行了barrier()，那么该工作组中其他工作项必须都要执行相同的barrier()，才能从当前节点继续往下执行（这种同步方式也称作*rendezvous*，即会合）。在进行化简操作时，这样做有两个用途。

- 在所有工作项从全局内存向局部内存复制数据的动作全部完成之前，确保任一工作项都不会开始进行化简操作，也确保了所有工作项都完成第*n*轮循环之前，任一工作项都不会开始第*n*+1轮循环。
- OpenCL只提供了宽松的内存一致性。这类似于2.2节介绍的Java内存模型的内存可见性——一个工作项对局部内存进行的修改并不保证对另一个工作项可见，除非处于某些同步点，比如同步屏障。所以，在每次循环结束时执行同步屏障，可以保证第*n*轮循环的结果对第*n*+1轮的所有工作项都可见。

运行内核

运行这个内核的方法与我们之前看到的类似——唯一的区别是如何创建局部缓存区：

DataParallelism/FindMinimumOneWorkGroup/find_minimum.c

```
CHECK_STATUS(clSetKernelArg(kernel, 2, sizeof(cl_float) * NUM_VALUES, NULL));
```

调用`clSetKernelArg()`时，将`arg_size`设置为缓存区的大小，将`arg_value`设置为NULL，这样就创建了一个局部缓存区。

现在已经可以用一个工作组进行化简操作了。不过工作组的大小是存在限制的（比如，我的Macbook Pro的GPU上不超过1024个元素）。下面再来看看如何使用多个工作组实现并行。

使用多个工作组进行化简操作

要使用多个工作组进行化简操作，只需要将输入数组进行切分并分别化简，如图7-8所示。

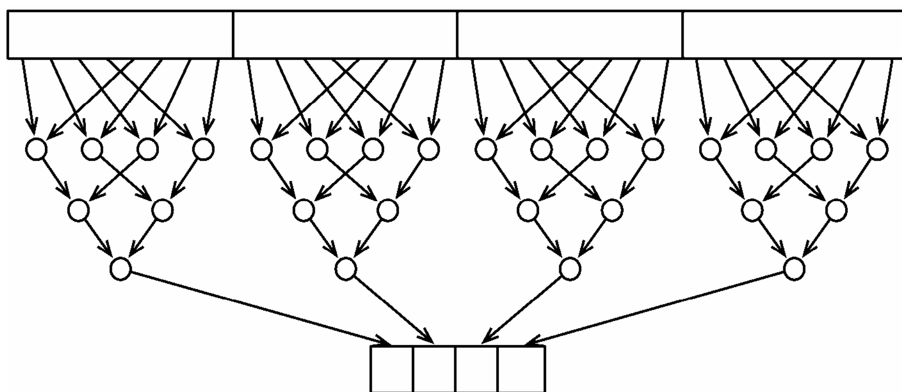


图7-8　使用多个工作组进行化简操作

举例说明，如果每个工作组一次处理64个值，那一个长度为*N*的数组会被化简成一个长度为*N*/64的数组。这个化简后的数组会再次进行化简，直到仅剩下一个值。

要做到这一点，就需要对内核做一些修改，这样才能处理工作组（工作组代表问题的一个部分）。为此，OpenCL会用局部ID识别工作项，这个ID是工作项在对应工作组中的ID，如图7-9所示。

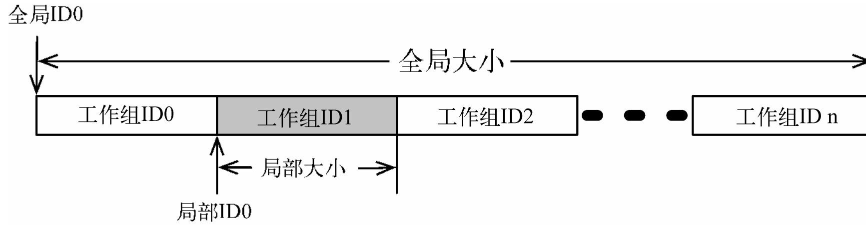


图7-9 工作组中的局部ID

下面这个内核使用了局部ID：

DataParallelism/FindMinimumMultipleWorkGroups/find_minimum.cl

```
__kernel void find_minimum(__global const float* values,
                           __global float* results,
                           __local float* scratch) {
➤ int i = get_local_id(0);
➤ int n = get_local_size(0);
➤ scratch[i] = values[get_global_id(0)];
  barrier(CLK_LOCAL_MEM_FENCE);
  for (int j = n / 2; j > 0; j /= 2) {
    if (i < j)
      scratch[i] = min(scratch[i], scratch[i + j]);
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (i == 0)
➤    results[get_group_id(0)] = scratch[0];
}
```

在之前`get_global_id()`和`get_global_size()`的位置，我们调用了`get_local_id()`和`get_local_size()`，其分别返回了工作项相对于工作组起始位置的ID和工作组的大小。在将值从全局内存复制到局部内存时，仍然调用`get_global_id()`，而向`results`数组存储结果时则调用`get_group_id()`。

还需要修改主机程序，创建合适数量的工作组：

DataParallelism/FindMinimumMultipleWorkGroups/find_minimum.c

```
size_t work_units[] = {NUM_VALUES};
size_t workgroup_size[] = {WORKGROUP_SIZE};
CHECK_STATUS(clEnqueueNDRangeKernel(queue, kernel, 1, NULL, work_units,
  workgroup_size, 0, NULL, NULL));
```

如果将`local_work_size`[①]设置为NULL，那么和往常一样，OpenCL平台将自主创建它认为合适数量的工作组。通过显式设置`local_work_size`的值，确保工作组的个数符合我们内核的要求（当然，最大个数是由设备决定的——关于查看这个限制的方法，参见前面的“查询设备信息”一节）

① `local_work_size`是`clEnqueueNDRangeKernel()`的第6个参数。——译者注

第二天总结

我们完成了第二天的学习。第三天将学习一个应用的例子，其使用OpenCL实现物理仿真，并与OpenGL集成来显示结果。

第二天我们学到了什么

OpenCL定义了一些用于抽象底层硬件细节的概念，包括平台、执行和内存模型。

- 工作项是在处理元件中执行的。
- 多个处理元件构成了计算单元。
- 在同一个计算单元中执行的一组工作项构成工作组。
- 一个工作组中的工作项之间通过局部内存进行通信，利用同步屏障进行数据同步，保证一致性。

第二天自习

查找

- 默认情况下，命令队列是按序执行命令的。如何进行乱序执行？
- 什么是事件等待列表（event wait list）？对于一个被发往无序命令队列的命令，如何利用事件等待列表来限制其执行的时机？
- `clEnqueueBarrier()`是做什么用的？同步屏障适用于何种场景？事件等待列表又适用于何种场景？

实践

- 修改化简数组的例子，使其支持任意个元素，而不是仅支持2的乘方个元素。
- 修改化简数组的例子，使其支持多个设备。如果只有一个支持OpenCL的设备，也可以使用CPU，或者用`clCreateSubDevices()`对GPU进行分区。你需要为每个设备创建一个命令队列，并将问题切分，使得一部分工作项在一个设备上执行，而另一部分工作项在另一个设备上执行，并保证命令队列之间的同步。
- 今天演示的化简算法非常简单。在互联网上搜索一下，你会发现有很多方法都可以改进化简的效率。在你的设备上能让化简变得多快？这些在GPU上适用的优化方法是否同样适用于CPU？

7.4　第三天：OpenCL 和 OpenGL——全部在 GPU 上运行

今天我们将完成一个完整的OpenCL应用，其实现一个物理仿真过程，并将结果可视化。在这个过程中，不仅会学习如何创建一个进行并行仿真的内核，还会学习如何将OpenCL和OpenGL集成起来，将整个过程都放在GPU上，以减少在不同缓存区之间复制数据的开销。

水波纹

今天的物理仿真的主题是水波纹。虽然这次仿真不会十分精细，但作为游戏中的一个效果应该足够了——比如模拟雨中的湖面。

LWJGL

本例中，我们将放弃使用C语言，转而使用Java及其LWJGL（LightWeight Java Graphics Library）库[1]，这样能更容易地创建跨平台的GUI。

LWJGL包装了OpenCL和OpenGL，Java程序可以通过它来使用OpenGL和OpenCL的C API。OpenGL从名称上看就与OpenCL联系紧密，特别是运行在GPU上的OpenCL内核可以直接使用OpenGL的缓存区。

使用LWJGL编写的OpenCL代码与之前C语言版本的代码十分类似。比如，下面的代码用于初始化OpenCL上下文、队列以及内核：

DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java

```
CL.create();
CLPlatform platform = CLPlatform.getPlatforms().get(0);
List<CLDevice> devices = platform.getDevices(CL_DEVICE_TYPE_GPU);
CLContext context = CLContext.create(platform, devices, null, drawable, null);
CLCommandQueue queue = clCreateCommandQueue(context, devices.get(0), 0, null);

CLProgram program =
  clCreateProgramWithSource(context, loadSource("zoom.cl"), null);
Util.checkCLError(clBuildProgram(program, devices.get(0), "", null));
CLKernel kernel = clCreateKernel(program, "zoom", null);
```

可以看出，这段代码的方法名和参数都像极了C语言版本的代码。为了填平语言之间的差异，比如Java中是没有指针的，还是需要修改少量代码。一般来说，是可以将C语言写成的OpenGL主机程序通过LWJGL翻译成Java版本的。

用 OpenGL 显示网格

我们并不打算用过多的篇幅来讨论OpenGL元素。不过确实需要花点时间，来解释本节的例子是如何显示用于表现水波纹的网格的，这样有利于阅读后面的OpenCL代码。

OpenGL的3D场景是由三角形构成的。在本例中，将三角形排列成如图7-10所示的网格。

可以用两个部分来描述每个三角形的位置：一个*顶点缓存区*（vertex buffer），其是顶点（在3D空间中的位置）的集合；一个*索引缓存区*（index buffer），其定义了如何用顶点来绘制三角形。

---

① http://www.lwjgl.org

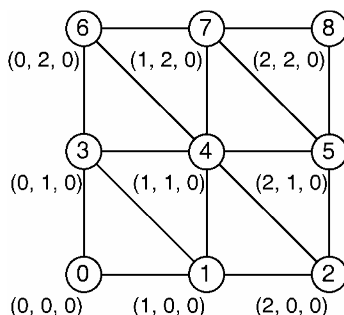


图7-10　三角形排列成的网格

在图7-10中，顶点0的位置是 (0, 0, 0)，顶点1的位置是 (1, 0, 0)，顶点2的位置是 (2, 0, 0)，以此类推。对应的顶点缓存区是[0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 1, 0, 1, 1, 0, ...]。

对于索引缓存区，第一个三角形使用了顶点0、1、3；第二个三角形使用了顶点1、3、4；第三个使用了顶点1、2、4；以此类推。对应的索引缓存区定义了一个三角形带（triangle strip），其通过三个顶点定义了第一个三角形，之后的每个三角形只需要额外定义一个点即可，如图7-11所示。

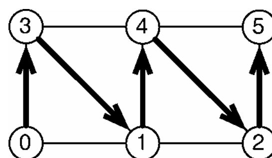


图7-11　三角形带

本书配套代码中定义了一个`Mesh`类，用来生成顶点缓存区和索引缓存区的初始值。下面的代码使用这个类构造了一个64 × 64的网格，其*x*轴和*y*轴的范围都是从−1.0到1.0：

DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java

```
Mesh mesh = new Mesh(2.0f, 2.0f, 64, 64);
```

其*z*轴的值始终为0——稍后涉及水波纹动画时会使用非0值。

生成的数据会被复制到OpenGL缓存区中：

DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java

```
int vertexBuffer = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
glBufferData(GL_ARRAY_BUFFER, mesh.vertices, GL_DYNAMIC_DRAW);

int indexBuffer = glGenBuffers();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, mesh.indices, GL_STATIC_DRAW);
```

这段代码首先使用glGenBuffers()为每个缓存区分配ID，然后用glBindBuffer()将其绑定到一个目标上，并用glBufferData()为其设置初始值。索引缓存区使用了GL_STATIC_DRAW标记，意味着它是不变的（静态的）；而顶点缓存区使用了GL_DYNAMIC_DRAW标记，意味着其会在动画帧之间发生改变。

在实现水波纹的代码之前，先尝试一个简单的例子——创建一个简单的内核，让其随着时间而放大网格的面积。

从 OpenCL 内核访问 OpenGL 缓存区

下面这个内核实现了放大动作：

DataParallelism/Zoom/src/main/resources/zoom.cl

```
__kernel void zoom(__global float* vertices) {

  unsigned int id = get_global_id(0);
    vertices[id] *= 1.01;
}
```

其参数是一个顶点缓存区，负责将该缓存区的每个元素都乘以1.01，每次调用该内核会将网格的面积放大1%。

为了能将顶点缓存区传给内核，要创建一个OpenCL缓存区来引用这个顶点缓存区：

DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java

```
CLMem vertexBufferCL =
  clCreateFromGLBuffer(context, CL_MEM_READ_WRITE, vertexBuffer, null);
```

在负责绘制的主循环代码中可以使用这个缓存区对象：

DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java

```
while (!Display.isCloseRequested()) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  glTranslatef(0.0f, 0.0f, planeDistance);
  glDrawElements(GL_TRIANGLE_STRIP, mesh.indexCount, GL_UNSIGNED_SHORT, 0);

  Display.update();

➤ Util.checkCLError(clEnqueueAcquireGLObjects(queue, vertexBufferCL, null, null));
➤ kernel.setArg(0, vertexBufferCL);
➤ clEnqueueNDRangeKernel(queue, kernel, 1, null, workSize, null, null, null);
➤ Util.checkCLError(clEnqueueReleaseGLObjects(queue, vertexBufferCL, null, null));
➤ clFinish(queue);
}
```

这段代码在OpenCL内核使用OpenGL缓存区前，首先调用了clEnqueueAcquireGLObjects()进行申请。然后将这个缓存区作为内核的一个参数，并像以前一样调用了clEnqueueNDR

angeKernel()。最后通过clEnqueueReleaseGLObjects()来释放缓存区，并调用clFinish()来等待发往命令队列的命令运行完成。

运行这段代码，可以看到一个网格刚开始很小，但快速地被放大，最终成为一个充满整个屏幕的三角形。

我们已经完成了一个简单的动画，其将OpenGL和OpenCL结合在一起。接下来可以实现更复杂的内核来仿真水波纹了。

仿真水波纹

现在要开始仿真水波纹的扩散效果了。每个波纹由两个参数来定义：一个扩散的中心点（网格中的2D点）和一个开始扩散时间。传给内核的参数包括一个指向OpenGL顶点缓存区的指针、一个含有扩散中心点的数组以及一个时间数组（时间单位是毫秒）：

DataParallelism/Ripple/src/main/resources/ripple.cl

```
Line 1 #define AMPLITUDE 0.1
     - #define FREQUENCY 10.0
     - #define SPEED 0.5
     - #define WAVE_PACKET 50.0
     5 #define DECAY_RATE 2.0
     - __kernel void ripple(__global float* vertices,
     -                      __global float* centers,
     -                      __global long* times,
     -                      int num_centers,
    10                      long now) {
     - unsigned int id = get_global_id(0);
     - unsigned int offset = id * 3;
     - float x = vertices[offset];
     - float y = vertices[offset + 1];
    15 float z = 0.0;
     -
     - for (int i = 0; i < num_centers; ++i) {
     -   if (times[i] != 0) {
     -     float dx = x - centers[i * 2];
    20     float dy = y - centers[i * 2 + 1];
     -     float d = sqrt(dx * dx + dy * dy);
     -     float elapsed = (now - times[i]) / 1000.0;
     -     float r = elapsed * SPEED;
     -     float delta = r - d;
    25     z += AMPLITUDE *
     -       exp(-DECAY_RATE * r * r) *
     -       exp(-WAVE_PACKET * delta * delta) *
     -       cos(FREQUENCY * M_PI_F * delta);
     -     }
    30   }
     -   vertices[offset + 2] = z;
     - }
```

上面的代码先获取了当前工作项的顶点在x轴和y轴的位置（第13、14行）。循环（第17~30行）是用于计算在z轴的位置，之后将这个位置写回顶点缓存区（第31行）。

在循环中，依次检查了每一个开始扩散时间非0的水波纹。对于每一个这样的水波纹，首先计算当前工作项的顶点到水波纹中心的距离d（第21行），然后计算水波纹扩散的半径r（第23行），以及当前顶点到波纹的距离 δ（第24行），如图7-12所示。

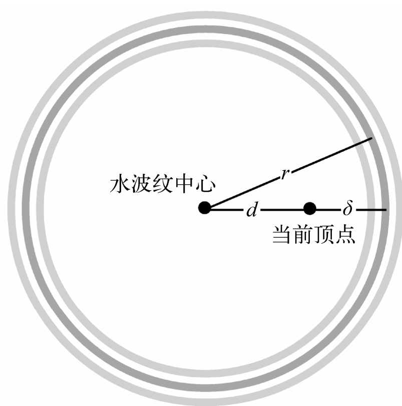


图7-12　水波纹的参量

综合r和 δ 计算得到z：

$$z = Ae^{-Dr^2}e^{-W\delta^2}\cos(F\pi\delta)$$

其中，A、D、W、F是常量，分别表示波纹的幅度、幅度随着扩散的衰减程度、波纹的宽度和波纹的频率。

最后还需要修改一下主机程序，创建缓存区：

DataParallelism/Ripple/src/main/java/com/paulbutcher/Ripple.java

```
int numCenters = 16;
int currentCenter = 0;
FloatBuffer centers = BufferUtils.createFloatBuffer(numCenters * 2);
centers.put(new float[numCenters * 2]);
centers.flip();
LongBuffer times = BufferUtils.createLongBuffer(numCenters);
times.put(new long[numCenters]);
times.flip();

CLMem centersBuffer =
  clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,centers, null);
CLMem timesBuffer =
  clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, times, null);
```

并在点击鼠标时触发新的水波纹：

DataParallelism/Ripple/src/main/java/com/paulbutcher/Ripple.java

```
while (Mouse.next()) {
  if (Mouse.getEventButtonState()) {
    float x = ((float)Mouse.getEventX() / Display.getWidth()) * 2 - 1;
    float y = ((float)Mouse.getEventY() / Display.getHeight()) * 2 - 1;

    FloatBuffer center = BufferUtils.createFloatBuffer(2);
    center.put(new float[] {x, y});
    center.flip();
    clEnqueueWriteBuffer(queue, centersBuffer, 0,
      currentCenter * 2 * FLOAT_SIZE, center, null, null);
    LongBuffer time = BufferUtils.createLongBuffer(1);
    time.put(System.currentTimeMillis());
    time.flip();

    clEnqueueWriteBuffer(queue, timesBuffer, 0,
      currentCenter * LONG_SIZE, time, null, null);
    currentCenter = (currentCenter + 1) % numCenters;
  }
}
```

编译并运行这段代码，多次点击网格，将会看到如图7-13所示的景象。

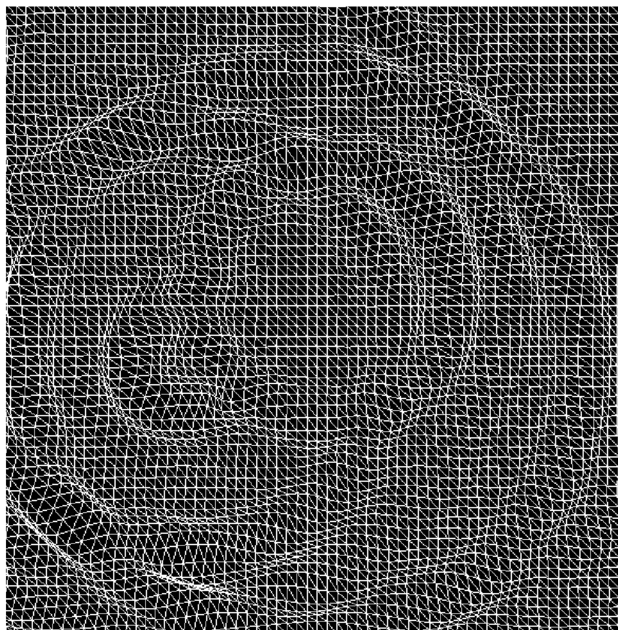


图7-13 水波纹

我们成功了——在GPU上完成了一次物理仿真，通过并行计算不仅进行了仿真，还对结果进

行了3D可视化。仿真和可视化需要的数据都在GPU上，不需要多余的数据复制。

第三天总结

我们完成了第三天的学习，同时也结束了用OpenCL在GPU上进行数据并行的讨论。

第三天我们学到了什么

在GPU上运行的OpenCL内核可以直接使用在同一个GPU上运行的OpenGL程序的缓存区。我们已经学习了以下内容：

- 在OpenCL中用`clCreateFromGLBuffer()`为一个OpenGL缓存区创建引用；
- 将OpenGL缓存区作为参数传给内核前，使用`clEnqueueAcquireGLObjects()`提出申请；
- 内核运行结束时，使用`clEnqueueReleaseGLObjects()`释放缓存区。

第三天自习

查找

- 什么是图像对象（image object）？它和OpenCL的缓存区对象有什么区别？当不需要与OpenGL交互时，图像对象是否适用？
- 什么是采样器对象（sampler object）？它适用于解决什么样的问题？
- 什么是原子函数（atomic function）？什么场景下会用原子函数替代同步屏障？

实践

- 在不使用原子函数的情况下，创建一个内核，其接受一个整数缓存区（整数范围为0~32），统计缓存区中每个整数的出现次数并形成统计直方图。如果将整数范围调整为0~1024，方案应如何修改？
- 使用原子函数创建内核再次解决上面的问题，并比较这两个方案。

7.5　复习

出于某些原因，在一些关于并行的主流讨论中数据并行常被忽略。不过通过本章的学习，我们已经了解到数据并行是一种非常强力的工具，可以大大改善代码的效率，所有程序员都应该将其收录到自己的工具箱中。

优点

数据并行非常适用于处理大量数值数据，尤其适合于科学计算、工程计算以及仿真领域，比如流体力学、有限元分析、*N*体模拟、模拟退火、蚁群优化、神经网络等。

GPU不仅是强大的数据并行处理器，在能耗方面也表现出众，比传统的CPU有更优秀的GFLOPS/watt①指标。世界上最快的超级计算机都广泛使用GPU或专用数据并行协处理器②，其中能耗指标低是一个重要的原因。

缺点

数据并行编程，更准确地说是GPGPU编程，在其适用的领域内所向披靡。但它并不适用于所有问题领域。值得一提的是，虽然用数据并行可以解决一些非数值问题（比如自然语言处理），但这样做并不容易——现今的工具集绝大多数关注的是数值处理。

对OpenCL内核进行调优是一个技术活，理解了底层架构的细节才能有效地进行调优。如果要写出高效的跨平台的代码，就会变得异常复杂。在解决某些问题时，从主机往设备上复制数据会消耗大量的时间，这会减弱甚至抵消我们从并行计算中获得的收益。

其他语言

GPGPU框架还包括CUDA③、DirectCompute④以及RenderScript Computation⑤。

结语

GPGPU编程是小规模应用数据并行技术的例子——所谓小规模，指的是程序运行在一台计算机上。下一章我们将学习Lambda架构，使用它可以大规模（跨越多台计算机）应用数据并行技术。

① GFLOPS是十亿次浮点运算/秒（giga floating-point operations per second）的缩写。GFLOPS/watt是用于衡量GPU效能比的单位。——译者注

② http://www.top500.org/lists/2013/06/

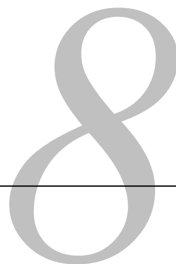
③ http://www.nvidia.com/object/cuda_home_new.html

④ http://msdn.com/directx

⑤ http://developer.android.com/guide/topics/renderscript/compute.html

第 8 章

Lambda架构



如果需要将一大批货物从国家的一端运往另一端，18轮的大卡车是不二之选。如果仅运送一个快递包裹，大卡车就不太适用了，因此综合性的航运公司也会使用一些小货车进行本地的货物收发。

Lambda架构采用了类似的方法，既使用了可以进行大规模数据批处理的MapReduce技术，也使用了可以快速处理数据并及时反馈的流处理技术，这样的混搭能够为大数据问题提供扩展性、响应性和容错性都很优秀的解决方案。

8.1 并行计算搞定大数据

近年来，大数据时代的到来为数据处理领域带来了巨大的变化。不同于传统数据处理，大数据领域广泛使用了并行计算——只要有足够的计算资源就可以处理TB级别的数据。Lambda架构是一种大数据处理技术，源于Nathan Marz在BackType和Twitter的经验总结并由此推广开来。

与上一章讨论的GPGPU编程类似，Lambda架构也使用了数据并行技术。与GPGPU编程不同，Lambda架构是站在大规模场景的角度来解决问题的，它可以将数据和计算分布到几十台或几百台机器构成的集群上进行。这种技术不但解决了之前因为规模庞大而无法解决的难题，还可以构建出对硬件错误和人为错误进行容错的系统。

Lambda架构包含了很多内容，本章只侧重于其并发和分布式特性（如需要深入学习，推荐阅读Nathan的著作*Big Data* [MW14]）。对于Lambda架构中的诸多组件，本书将侧重介绍两个主要的：批处理层（Batch Layer）和加速层（Speed Layer），如图8-1所示。

批处理层使用MapReduce这类批处理技术从历史数据中对批处理视图进行预计算。这种计算效率很高但延迟也很高，所以又增加了一个加速层，使用流处理等低延迟技术从接收到的新数据中计算实时视图。合并这两种视图，就可以获得最终的计算结果。

本书写到现在，Lambda架构是最复杂的专题。它以很多其他技术为基石，其中最重要的就是MapReduce。第一天，我们将只学习MapReduce技术，而忽略其使用的场景。第二天，首先学习传统数据系统中的问题，然后学习在Lambda架构的批处理层如何使用MapReduce技术解决这些问题。第三天，学习流处理技术，以及如何使用这项技术构造加速层，这样就可以一窥Lambda

架构的全貌了。

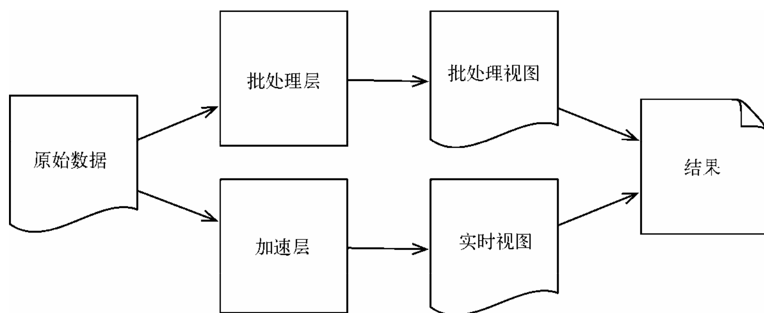


图8-1　批处理层和加速层

8.2　第一天：MapReduce

MapReduce是一个多义的术语。其可以指代一类算法，这类算法分为两个步骤：对一个数据结构首先进行映射（map）操作，然后进行化简（reduce）操作。之前的词频统计的函数式版本正是这样的例子（`frequencies`就是用`reduce`函数实现的）。我们在3.3节中讨论过，将算法拆成映射和化简两步的一个好处是易于并行化。

MapReduce还可以指代一类系统——这类系统使用了上面的算法，将计算过程高效地分布到一个集群上。这类系统不仅可以将数据和数据处理分布到集群的多台计算机上，还可以在一台或多台计算机崩溃时继续正常运转。

当MapReduce指代一类系统时，可以说它是Google发明的[①]。除了Google，最流行的MapReduce框架是Hadoop[②]。

今天将结合前面的Wikipedia词频统计的例子，用Hadoop实现一个使用MapReduce的并行版本。Hadoop支持多种编程语言——我们选用Java。



小乔爱问：

怎么用了这么个名称?

对于“Lambda 架构”这个名称有多种推测。我认为最好的解释来自于 Lambda 架构之父 Nathan Marz[a]：

Lambda 架构源自于它与函数式编程的相似性。从本质上说，Lambda 架构是将计算函数

① http://research.google.com/archive/mapreduce.html

② http://hadoop.apache.org

施加于大量数据的一种通用方法。（原文：The name is due to the deep similarities between the architecture and functional programming. At the most fundamental level, the Lambda Architecture is a general way to compute functions on all your data at once）

a. http://www.manning-sandbox.com/message.jspa?messageID=126599

可行性

开发和调试MapReduce程序的起点是在本地运行Hadoop。要在一个集群上运行Hadoop在过去是很痛苦的——不是每位读者都有足够的闲置计算机来组建一个集群，而且就算能组建一个集群，安装、配置和维护Hadoop集群需要大量的时间和精力。

幸运的是，云计算提供了按需使用、计时收费的虚拟机服务，从而大大改善了这一状况。而且许多云计算供应商直接提供了Hadoop集群的管理服务，大大简化了集群的配置和维护。

我们将使用Amazon Elastic MapReduce（EMR）服务来运行本章的例子[①]。之后都将在EMR上进行集群的启动、停止、复制数据等操作，其背后的原理同样适用于所有Hadoop集群。

为了运行本章的例子，需要注册一个Amazon AWS账号，并安装AWS和EMR命令行工具[②,③]。



小乔爱问：

如何搞定Hadoop的版本？

Hadoop 一直执着地使用一套混乱的版本号体系，本书撰写时其活跃的版本号就有 0.20.x、1.x、0.22.x、0.23.x、2.0.x、2.1.x 和 2.2.x。这些版本支持两套 API，一套是“旧”的 API（`org.apache.hadoop.mapred` 包），另一套是“新”的 API（`org.apache.hadoop.mapreduce` 包）。

另外，不同的 Hadoop 发行版会打包 Hadoop 某个版本和一系列的第三方组件[a,b,c]。

本章的例子都会使用“新”的 API，并在 Amazon 3.0.2 AMI（Hadoop 2.2.0[d]）上测试通过。

a. http://hortonworks.com
b. http://www.cloudera.com
c. http://www.mapr.com
d. http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-plan-hadoop-version.html

① http://aws.amazon.com/elasticmapreduce/
② http://aws.amazon.com/cli/
③ http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-reference.html

Hadoop基础

Hadoop就是用来处理大量数据的工具。如果你的数据不是以吉字节或者更大的单位来度量，那就不适合使用Hadoop。Hadoop的效率源自于它将数据分块后分别交给多台计算机进行处理。

我们很容易猜到，一个MapReduce任务由两种主要的组件构成：mapper和reducer①。mapper负责将某种输入格式（通常是文本）映射为许多键值对。reducer负责将这些键值对转换成最终的输出格式（通常也是一系列键值对）。mapper和reducer可以分布在很多不同的计算机上（它们的数目不必相同），如图8-2所示。

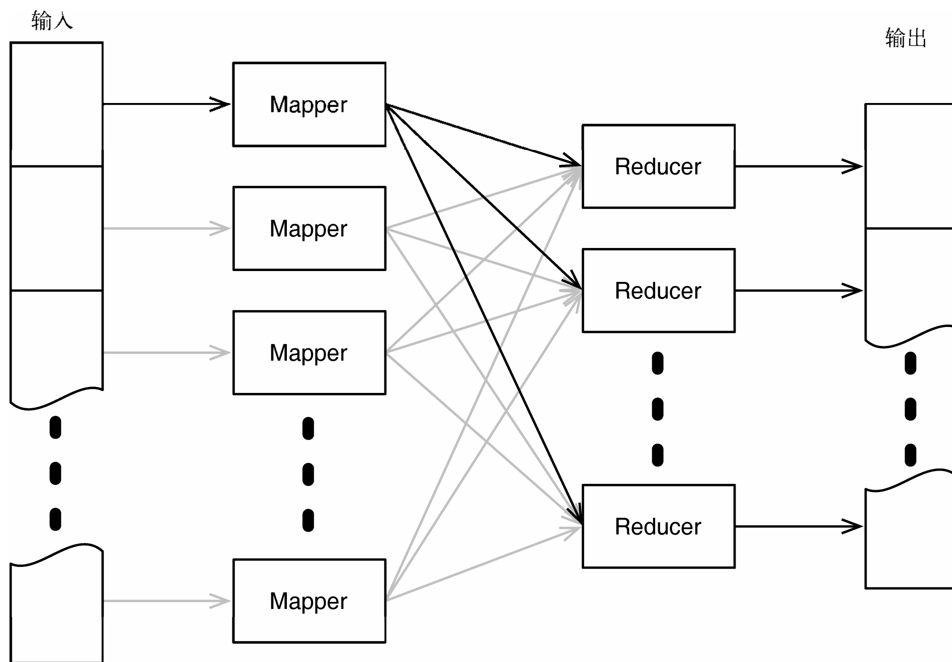


图8-2　Hadoop数据流

输入通常由一个或多个大文本文件构成。Hadoop对这些文件进行分片（每一片的大小是可配置的，通常为64 MB），并将每个分片发送给一个mapper。mapper将输出一系列键值对，Hadoop再将这些键值对发送给reducer。

一个mapper产生的键值对可以发送给多个reducer。键值对的键决定了哪个reducer会接受这个键值对——Hadoop确保具有相同键的键值对（无论是由哪个mapper产生的）都会发送给同一个reducer处理。这个阶段通常被称为洗牌阶段（shuffle phase）。

① 之前的章节将mapper和reducer分别译为“映射器”和“化简器”，本章中为了和Hadoop的类名对应，保留英文名称。——译者注

Hadoop为每个键调用一次reducer，并传入所有与该键对应的值。reducer将这些值合并，再生成最终输出结果（通常是键值对，也可以不是）。

理论略显枯燥——之前介绍过Wikipedia词频统计的例子，现在来实现它的Hadoop版本。

词频统计的Hadoop版本

为了让起步更平稳，我们先将需求简化为：统计几个文本文件中的词频（之后会将需求扩展到统计Wikipedia dump的词频）。

本例的mapper每次会处理一行文本，将其切分为单词，再用键值对来描述每个单词。键值对的键是单词本身，而值则是常数1。对于每个单词，本例的reducer会对相关的所有键值对的值进行求和，并生成一个结果键值对，该键值对的值是这个单词在整个输入中出现的次数。如图8-3所示。

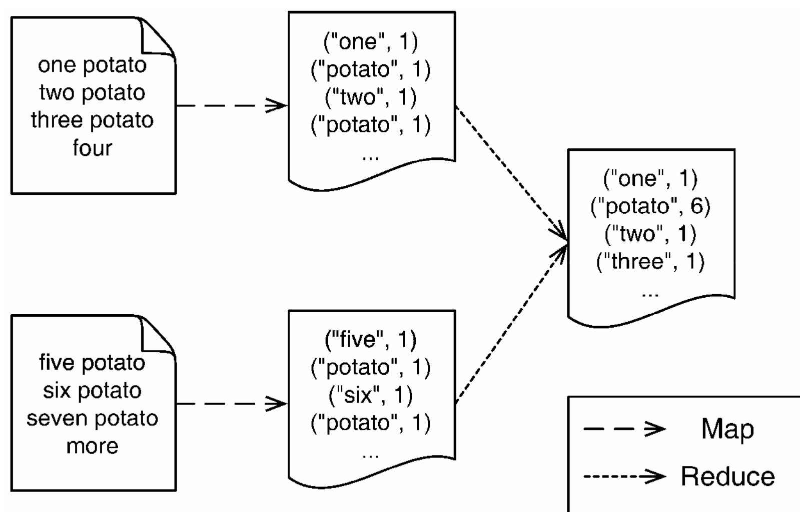


图8-3　词频统计的Hadoop版本

Mapper

我们的Map继承了Hadoop的Mapper类，其接受四个类型参数：输入的键类型、输入的值类型、输出的键类型、输出的值类型：

LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/WordCount.java

```
Line 1 public static class Map extends Mapper<Object, Text, Text, IntWritable> {
     -   private final static IntWritable one = new IntWritable(1);
     -
     -   public void map(Object key, Text value, Context context)
     5     throws IOException, InterruptedException {
```

```
-
-     String line = value.toString();
-     Iterable<String> words = new Words(line);
-     for (String word: words)
10      context.write(new Text(word), one);
-   }
- }
```

Hadoop表示输入和输出时需要使用自己的数据类型（不能直接使用`String`或者`Integer`）。mapper要处理的是文本数据，而不是键值对，因此不需要输入的键类型（用`Object`替代），而输入的值类型是`Text`。输出的键类型是`Text`，值类型是`IntWritable`。

每处理一行输入文本都要调用一次`map()`方法，对输入的行进行切分。首先将输入的行转换为Java的`String`类型（第7行），然后将字符串切分为单词（第8行），最后遍历所有单词，为每一个单词生成相应的键值对，其键是单词本身，其值是常数`1`（第10行）。

Reducer

我们的`Reduce`继承了Hadoop的`Reducer`类，与`Mapper`类似，其参数也描述了输入和输出的键值类型（本例中键类型都是`Text`，值类型都是`IntWritable`）：

LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/WordCount.java

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val: values)
      sum += val.get();
    context.write(key, new IntWritable(sum));
  }
}
```

对于每个键，都会调用一次`reduce()`方法，`values`是这个键对应的所有值的集合。`reduce()`方法对这些值进行求和，并产生描述某个单词出现总数的键值对。

现在已经得到了一个mapper和一个reducer，剩下的任务就是创建一个driver，这样Hadoop才知道如何让这几个部分运转起来。

Driver

我们的driver是一个Hadoop的`Tool`，实现了`run()`方法：

LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/WordCount.java

```
Line 1 public class WordCount extends Configured implements Tool {
-
-        public int run(String[] args) throws Exception {
-          Configuration conf = getConf();
5          Job job = Job.getInstance(conf, "wordcount");
-          job.setJarByClass(WordCount.class);
```

```
-     job.setMapperClass(Map.class);
-     job.setReducerClass(Reduce.class);
-     job.setOutputKeyClass(Text.class);
10    job.setOutputValueClass(IntWritable.class);
-     FileInputFormat.addInputPath(job, new Path(args[0]));
-     FileOutputFormat.setOutputPath(job, new Path(args[1]));
-     boolean success = job.waitForCompletion(true);
-     return success ? 0 : 1;
15  }
-
-   public static void main(String[] args) throws Exception {
-     int res = ToolRunner.run(new Configuration(), new WordCount(), args);
-     System.exit(res);
20  }
- }
```

这段代码主要是公式化地告诉Hadoop我们要做什么。首先，第7行和第8行设置了mapper和reducer的类，第9行和第10行设置了输出的键类型和值类型。这里不需要设置输入的键类型和值类型，因为默认情况下Hadoop认为我们处理的是文本文件。也不需要分别设置mapper输出的键/值类型和reducer输入的键/值类型，因为默认情况下Hadoop认为mapper的输出和reducer的输入具有相同的键/值类型。

然后，第11行和第12行告知Hadoop如何获得输入数据以及如何输出结果。最后，第13行启动任务并等待任务结束。

现在已经完成了一个完整的Hadoop任务，可以输入一些数据运行一下了。

在本地运行

先来尝试在本地运行Hadoop任务。在本地运行时程序无法并行执行，也无法容错，不过可以用最小代价来验证一下程序是否运行正常，而不需要将程序部署到集群上再进行验证。

我们需要一些文本作为输入数据。input文件夹中有两个文本文件，包括即将进行分析的文本：

LambdaArchitecture/WordCount/input/file1.txt

```
one potato two potato three potato four
```

LambdaArchitecture/WordCount/input/file2.txt

```
five potato six potato seven potato more
```

虽然输入数据很短，显然不够吉字节级别，不过足够用来检验代码正确性。要对这两个文本文件进行词频统计，可以用mvn package命令进行编译，再调用下面的命令在本地启动Hadoop实例：

```
$ hadoop jar target/wordcount-1.0-jar-with-dependencies.jar input output
```

当Hadoop运行完成，就可以看到一个新文件夹output，其包括两个文件——_SUCCESS和

part-r-00000。_SUCCESS是一个空文件，只是告诉我们任务运行成功。part-r-00000的内容如下：

```
five 1
four 1
more 1
one 1
potato 6
seven 1
six 1
three 1
two 1
```

我们已经在小数据规模的情况下，在本地验证了任务的正确性，现在需要在真正的集群上运行这个任务，并处理更多的输入。



小乔爱问：

结果一定是排序好的吗?

你也许注意到了输出的结果是按键的字符序进行排序的。Hadoop 在键值对传给 reducer 前会对键进行排序，这在一些场景下会有帮助。

但需要小心。虽然键值对传给 reducer 前会对键进行排序，但稍后将学习到，默认情况下 reducer 之间是没有顺序的。partitioner 组件可以用于控制这一行为，但本书不再详述。

在Amazon EMR上运行

要在Amazon Elastic MapReduce上运行一个Hadoop任务的步骤比较复杂。本书不会深入讨论EMR，而仅介绍一些必要的细节。

输入和输出

EMR默认都是对Amazon S3[①]进行输入和输出。包含代码的JAR包以及日志文件也会存储在S3上。

首先，创建一个包含若干文本文件的S3 bucket。由于Wikipedia dump是XML文件而不是文本文件，所以不太适用。本章的配套代码中有一个项目ExtractWikiText，可以从Wikipedia dump中提取需要的文本。然后，将这些文本上传到S3 bucket中。代码编译生成的JAR包需要上传到另一个S3 bucket中。

① http://aws.amazon.com/s3/

向S3上传大文件

如果在上传大文件时，你的宽带像我的一样不够“宽”，可以考虑创建一个临时的Amazon EC2实例，利用这个实例下载Wikipedia dump、提取文本并将文本上传到S3上。毫无疑问，EC2和S3之间有足够的带宽。

创建一个集群

创建EMR的集群有许多方法——本书使用的是命令行工具`elastic-mapreduce`：

```
$ elastic-mapreduce --create --name wordcount --num-instances 11 \
--master-instance-type m1.large --slave-instance-type m1.large \
--ami-version 3.0.2 --jar s3://pb7con-lambda/wordcount.jar \
--arg s3://pb7con-wikipedia/text --arg s3://pb7con-wikipedia/counts
Created job flow j-2LSRGPBSR79ZV
```

上面的命令创建了一个名为`wordcount`的集群，其含有11个实例（1主10备），每个实例都是`m1.large`类型，并运行在3.0.2 AMI[①]上。最后的几个参数分别是JAR包在S3上的位置、输入数据在S3上的位置和输出数据在S3上的位置。

监控

创建集群时命令行返回了一个job flow的ID，我们可以用这个ID建立SSH连接，连接到刚才创建的集群：

```
$ elastic-mapreduce --jobflow j-2LSRGPBSR79ZV --ssh
```

现在已经处于主实例上的命令行中，通过查看日志文件可以对任务的进度进行监控：

```
$ tail -f /mnt/var/log/hadoop/steps/1/syslog
INFO org.apache.hadoop.mapreduce.Job (main): map 0% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 1% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 2% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 3% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 4% reduce 0%
```

检查结果

在我的测试中，对Wikipedia进行词频统计需要1小时多一点。运行完成后，可以在相应的S3 bucket中看到很多文件：

```
part-r-00000
part-r-00001
part-r-00002
```

① http://aws.amazon.com/ec2/instance-types/

```
⋮
part-r-00028
```

这些文件作为一个整体包含了所有结果。在每个结果分块中，结果是排序的，但整体上不是排序的（参见“小乔爱问：结果一定是排序好的吗？”）。

现在已经可以统计文本文件中的词频了，不过最好能直接统计Wikipedia dump的词频。下面来看看怎么做。

处理XML

XML文件其实只是对结构有要求的文本文件，所以我们很容易想到用处理文本文件的方式来处理XML文件。但这是行不通的，原因是Hadoop默认根据换行符对文件进行分片，而这可能会错误地切分XML标签。

虽然Hadoop默认没有提供针对XML的分片器，但利用另一个Apache项目Mahout[1]提供的`XmlInputFormat`[2]可以达到目的。为了使用`XmlInputFormat`，需要对driver进行一些修改：

LambdaArchitecture/WordCountXml/src/main/java/com/paulbutcher/WordCount.java

```
Line 1 public int run(String[] args) throws Exception {
     -   Configuration conf = getConf();
     -   conf.set("xmlinput.start", "<text");
     -   conf.set("xmlinput.end", "</text>");
     5
     -   Job job = Job.getInstance(conf, "wordcount");
     -   job.setJarByClass(WordCount.class);
     -   job.setInputFormatClass(XmlInputFormat.class);
     -   job.setMapperClass(Map.class);
    10   job.setCombinerClass(Reduce.class);
     -   job.setReducerClass(Reduce.class);
     -   job.setOutputKeyClass(Text.class);
     -   job.setOutputValueClass(IntWritable.class);
     -   FileInputFormat.addInputPath(job, new Path(args[0]));
    15   FileOutputFormat.setOutputPath(job, new Path(args[1]));
     -
     -   boolean success = job.waitForCompletion(true);
     -   return success ? 0 : 1;
     - }
```

在这段代码中，使用`setInputFormatClass()`（第8行）将`XmlInputFormat`设置为分片器，并且配置`xmlinput.start`和`xmlinput.end`（第3行和第4行）来告诉分片器我们关注的是哪个标签。

8

仔细查看`xmlinput.start`的值，你可能会觉得有点奇怪——这个值为`<text`，看上去是个

① http://mahout.apache.org

② https://github.com/apache/mahout/blob/trunk/integration/src/main/java/org/apache/mahout/text/wikipedia/XmlInputFormat.java

残缺的XML标签。`XmlInputFormat`对XML并不进行完整的解析，而只是匹配起始和终止的模式。由于`<text>`标签中可以设置属性，所以不能设置`xmlinput.start`为`<text>`，而需要设置成`<text`。

还需要修改一下mapper：

LambdaArchitecture/WordCountXml/src/main/java/com/paulbutcher/WordCount.java

```
private final static Pattern textPattern =
  Pattern.compile("^<text.*>(.*)</text>$", Pattern.DOTALL);
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException {

  String text = value.toString();
  Matcher matcher = textPattern.matcher(text);
  if (matcher.find()) {
    Iterable<String> words = new Words(matcher.group(1));
    for (String word: words)
      context.write(new Text(word), one);
  }
}
```

每个分片由匹配`xmlinput.start`和`xmlinput.end`标签之间的文本（包含被匹配的标签）构成。在进行统计之前，这段代码还用了一点正则表达式的技巧来去除`<text></text>`标签（防止`text`这个词的计数不准）。

你也许已经注意到driver中使用了`setCombinerClass()`（第10行）来设置combiner。combiner是一种优化手段，使键值对可以在发往reducer前进行合并（如图8-4所示）。我进行了一下测试，程序的运行时间从1个多小时下降到45分钟。

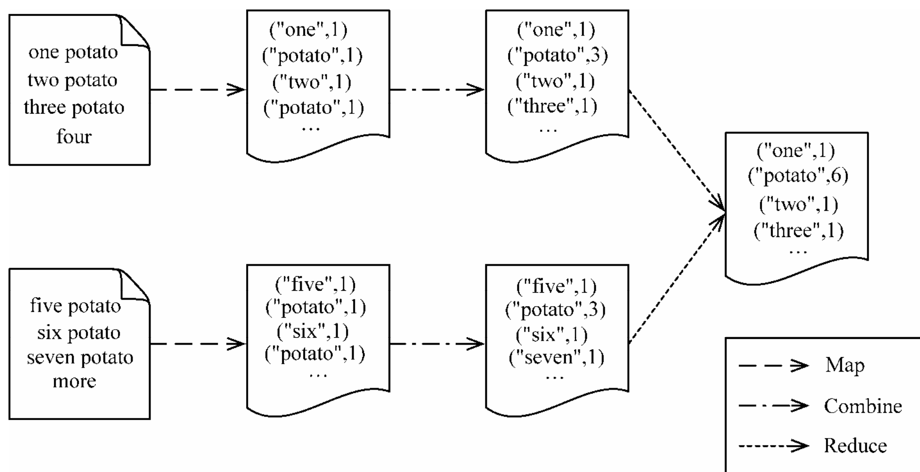


图8-4　使用combiner

在我们的场景中，reducer起的作用和combiner一样，但在其他场景中可能就需要单独的combiner。当设置了一个combiner时，Hadoop并不能保证一定会使用它，所以必须确定我们的算法不依赖于是否调用combiner，也不依赖于调用了多少次combiner。



小乔爱问：

Hadoop只有速度优势吗?

一个通常的误解是：Hadoop 的优势只有速度提升——比起使用一台计算机，Hadoop 可以在多台计算机上更快地处理海量的数据，这的确是个诱人的优势。但它还有其他优势。

- 当涉及数百台计算机构成的集群时，系统崩溃不再是一个“极少发生的风险”，而是一种“很有可能发生的必然”。如果一台计算机的崩溃就会引发整个系统崩溃，那么这个系统基本上没有实用价值。因此，Hadoop 天生就具有处理错误和从错误中恢复的能力。
- 与上一条相关，我们不仅要考虑将节点崩溃时正在处理的任务重新执行，还需要考虑当存储发生故障时如何保证数据不丢失。Hadoop 默认使用 Hadoop 分布式文件系统（HDFS），这个有容错能力的分布式文件系统可以在多个节点之间冗余数据。
- 涉及吉字节级别以上的数据时，就不能将所有中间数据或结果全部存放在内存中。Hadoop 在处理过程中将键值对存储在 HDFS 中，这样就可以不受内存限制，完成数据量非常大的任务。

综上所述，这些优点都是革命性的。本书只在本章中使用了完整的 Wikipedia dump 作为词频统计的输入数据，这并不是巧合——MapReduce 是本书介绍的唯一能处理这个量级数据的技术。

第一天总结

第一天的学习结束了。第二天我们将学习如何用Hadoop实现Lambda架构的批处理层。

第一天我们学到了什么

将一个问题拆分成一个映射操作和一个化简操作，使其更容易被并行化。MapReduce，在本章中使用的这个术语特指一个使用多台计算机的、由映射操作和化简操作构成的、高效且容错的分布式系统。Hadoop就是一个MapReduce系统，其可以做到：

- 将输入分配给多个mapper，每个mapper都会产生一些键值对；
- 这些键值对会被发送给reducer，产生最终的输出（通常也是一系列键值对）；
- 每个reducer对应的键是不同的，因此具有相同键的键值对都会发送给同一个reducer进行处理。

第一天自习

查找

- 阅读Hadoop streaming API的文档，通过Hadoop streaming可以使用其他语言创建MapReduce任务，比如Ruby、Python或Perl。
- 阅读Hadoop pipes API的文档，通过Hadoop pipes可以用C++创建MapReduce任务。
- 有很多基于Hadoop Java API的库，利用它们可以很容易地创建复杂的MapReduce任务。比如Cascading、Cascalog和Scalding。

实践

- 在词频统计程序运行时，尝试干掉集群中的一台计算机（不要干掉主节点——Hadoop不能处理主节点的崩溃）。检查整个过程中的日志，看看Hadoop如何重试故障节点上的任务。检查最终结果，其正确性不应受到故障的影响。
- 现在的词频统计程序能很好地完成本职工作，但如果想知道“Wikipedia最常用的100个词是什么？”，就需要改进一下程序。利用二级排序（Secondary Sort）可以获取全局排序的结果（网络上有许多文章介绍如何实现）。
- Top ten模式是解决“Wikipedia最常用的词”这个问题的另一种方法。利用这个模式尝试解决一下。
- 有些问题无法用单个MapReduce任务来解决——经常需要串联多个任务，前一个任务的输出是后一个任务的输入。以PageRank算法为例，创建一个Hadoop程序来计算每个Wikipedia页面的page rank。多少个迭代后结果才能达到稳定？

8.3　第二天：批处理层

昨天学习了如何用Hadoop在一个集群中进行并行计算。MapReduce适用于解决各种各样的问题，今天我们将学习在Lambda架构中如何使用MapReduce。

不过，在正式学习之前先来了解一下Lambda架构要解决的主要问题——传统数据系统有什么缺陷？

传统数据系统的缺陷

数据系统不是一个新概念——从计算机发明之初，数据库就一直负责存储和处理数据。传统数据库适用于一台计算机，但随着处理的数据量越来越大，数据库就必须使用多台计算机。

扩展性

利用某些技术（比如复制、分片等）可以将传统数据库扩展到多台计算机上，但随着计算机数量和查询数量的增加，应用这种方案会变得越来越困难。超过一定程度，增加计算机资源将无

法继续改善性能。

维护成本

维护一个跨多台计算机的数据库的成本是比较高的。如果要求维护时不能停机，那么维护将变得更加困难——比如对数据库进行重新分片。随着数据量和查询数量的增加，容错、备份、确保数据一致性等工作的难度都会呈几何级数增长。

复杂度

复制和分片通常要求应用层面提供一些支持——应用需要知道将查询发给哪一台计算机，以及应该更新哪一个数据分片（每个更新所对应的分片通常不一样，规则也比较复杂）。程序员习惯使用的许多特性（例如对事务的支持）在数据库分片后都无法使用。也就是说程序员必须显式处理失败的事务并进行重试。这些都增加了使用传统数据库的复杂度，也增加了出错的可能。

人为错误

讨论容错性时很容易被忽略的就是人为错误。许多数据故障不是由于存储故障引起的，而是由于管理员或开发人员的人为错误引起的。如果运气比较好，这类错误可以被快速定位，并通过还原备份来恢复，但不是所有错误都可以轻易解决。设想一下，如果有一个隐藏了几周的数据错误突然引发了大面积的崩溃，我们又该如何修复数据库呢？

有时，我们可以分析错误的影响范围，并写一个临时的脚本来修复数据库。有时，我们可以通过重放数据库日志（假设数据库日志记录了所有必要的信息）来回滚这个错误。有时，我们只能承认运气不佳。每次都依赖运气可不是一个好的长久之计。

报表与分析

传统数据库擅长于运营支持，即处理日常的业务数据。如果要处理历史数据，比如生成报表或进行数据分析，传统数据库的效率就比较低了。

典型的解决方案是在独立的数据仓库中用另一种格式来维护历史数据。数据从业务数据库向数据仓库的迁移过程就是著名的萃取（extract）、转置（transform）、加载（load）（简称ETL）。这种方案不仅复杂，而且需要准确预测将来我们需要什么信息。有时会碰到这种情况：由于缺乏必要的信息或者信息格式不对，无法生成所需报表或进行某些分析。

现在学习Lambda架构如何解决这些问题。Lambda架构不仅能处理现代应用中的大量数据，而且其使用也比较简单，可以从技术性故障和人为故障中恢复，并维护完整的历史数据，这样就可以在未来生成任何想要的报表，进行任何想要的分析。

永恒的真相

我们可以将信息分为两类——原始数据及（源于原始数据的）衍生信息。

以Wikipedia的页面为例，Wikipedia的页面是被持续更新的，也就是说今天看到的某个页面和昨天看到的同一个页面的内容可能是不同的。但是在Wikipedia的结构中，页面并不是原始数据——一个页面是由许多贡献者的若干次编辑记录构成的。这些编辑记录才是原始数据，而页面是其衍生信息。

另外，虽然页面每天都在变，但编辑记录是不变的。一旦贡献者进行了一次编辑，这个编辑记录就不会改变了。后续的编辑记录可能影响或回滚本条编辑产生的效果，也会影响到页面的内容，但编辑记录本身是不变的。

在任何数据系统中信息都可以这样分类。银行账户的余额是衍生信息，而账户的收入和支出是原始数据；Facebook的friend graph是衍生信息，而添加好友和删除好友的事件是原始数据。与Wikipedia的编辑记录类似，收入记录、支出记录、添加好友事件、删除好友事件都是不变的。

原始数据是永恒的真相，也是Lambda架构的基础。下一节我们将学习其如何利用原始数据来解决传统数据系统碰到的问题。



小乔爱问：

原始数据真的都是不变的？

乍看上去，有一些原始数据不大可能是永远不变的。比如用户的家庭住址？如果用户搬家了呢？

这类数据可以是不变的——我们只需要添加一个时间戳。以前我们的记录是 Charlotte lives at 22 Acacia Avenue，而添加时间戳后的记录是 On March 1, 1982, Charlotte lived at 22 Acacia Avenue。这样，无论将来发生什么，原始数据仍然是不变的。

数据还是原始的好

建议大家现在集中注意力。如前所述，不变性和并行计算是天作之合。

美好的设想

现在来做一个简短的设想。假如有一个无限快的计算机，可以在瞬间处理TB级别的数据。那么只需要保存原始数据而不需要保存衍生信息，因为在需要的时候可以由原始数据推导出衍生信息。

在这种情况下，由于数据是不变的，存储数据的成本大幅下降，就大大降低了传统数据库系统的复杂度。存储介质只需要让我们附加新的数据即可。由于数据被存储后就不可变了，那就不再需要那些精密的锁机制和事务机制了。

更进一步，当数据不可变时，多个线程可以并行地访问数据，而不用担心相互之间的作用。我们可以对数据进行复制，再对副本进行操作，而不用担心数据过期，所以在集群中分布地处理数据就变得非常容易。



小乔爱问：

删除数据该怎么处理？

有些情况下，我们有足够的理由删除原始数据。原因可能是数据已经不再使用了，也可能是审计或安全的因素（比如，数据保护法规可能要求数据存在一段时间后不再继续保存）。

这并不意味着我们之前说错了。尽管可以选择遗忘那些要被删除的原始数据，它们仍然是不变的。

当然，设想终归是设想，不过见识了 MapReduce 的威力之后，你会惊讶于我们是如此接近理想。

设想（几乎）变为现实

如果能够准确预测出未来会对原始数据进行怎样的查询，就可以预计算出一个批处理视图，这个视图包含这些查询将要返回的衍生信息，或者那些可以计算出这些衍生信息的数据。Lambda架构的批处理层就是用来计算这些批处理视图的。

批处理视图可以包含衍生信息，比如：假设要用一系列编辑记录来构建Wikipedia的页面——批处理视图将只包含从页面的编辑记录中计算得来的页面内容。

批处理视图也可以包含可以计算出衍生信息的数据，这类情况会稍微复杂一些，这也是今天的讨论重点。我们将用Hadoop来构造批处理视图，通过批处理视图可以查询某个Wikipedia贡献者在某一段时间内进行了多少次编辑。

Wikipedia贡献者

我们理想中的查询应该是这样的："Fred Bloggs在2012年6月5日下午3:15到2012年6月7日上午10:45之间进行了多少次编辑？"为了达到这个目的，就需要维护每个编辑记录的编辑时间并进行索引。如果这种查询是必要的，那我们的成本会比较高，不过实际上不需要如此细粒度的查询——按天来维护数据已经绰绰有余了。

所以批处理视图就会按天进行统计，如图8-5所示。

如果只需要查询几天的状况，那现在已经满足要求了，但如果要查询几个月的状况，就需要合并许多值（如果需要一年的数据，就需要统计365天的贡献次数）。如果能同时按天和按月维护

数据，就可以减少计算工作量，如图8-6所示。

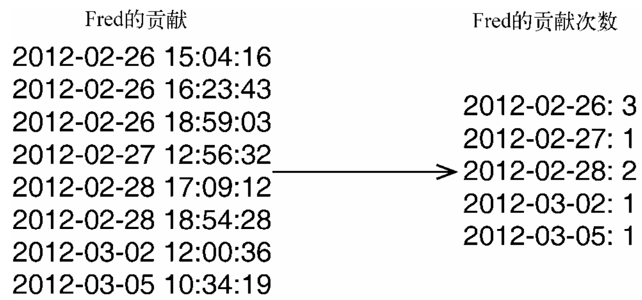


图8-5　按天统计的批处理视图

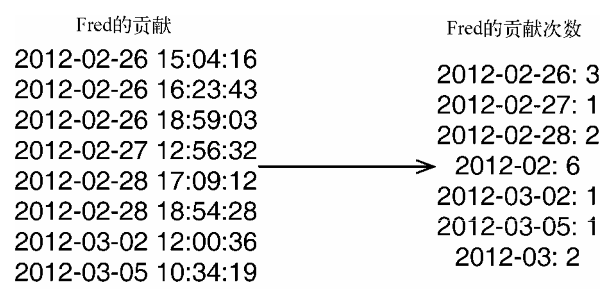


图8-6　按天和按月统计的批处理视图

计算一年中某用户的贡献数时，计算次数从365次降到了12次。通过增加或删减以天为单位的数据，就可以处理开始时间或结束时间不是整月的查询时间区段，如图8-7所示。

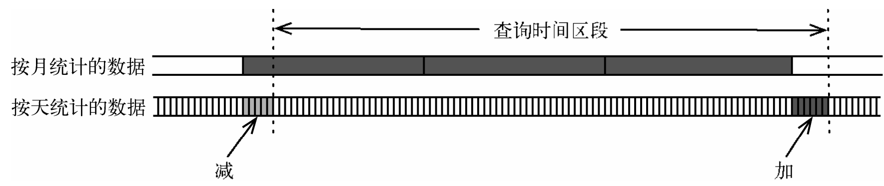


图8-7　查询某用户在一个时间区段内的贡献数

贡献者日志

遗憾的是，我们无法获得Wikipedia贡献者的feed。我们希望feed是如下格式的：

```
2012-09-01T14:18:13Z 123456789 1234 Fred Bloggs
2012-09-01T14:18:15Z 123456790 54321 John Doe
2012-09-01T14:18:16Z 123456791 6789 Paul Butcher
⋮
```

第一列是时间戳，第二列是贡献记录的标识符，第三列是贡献者的用户ID，第四列是贡献者的用户名。

Wikipedia虽然没有提供这样的feed，却提供了包含全部历史数据的周期性的XML dump[①]（我们需要其中的enwiki-latest-stub-meta-history）。

本章的配套代码中有一个项目ExtractWikiContributors，其可以将一个dump转化为上述feed格式的日志文件。

下一节我们将创建一个Hadoop任务，接受这些日志文件并生成批处理视图需要的数据。

计算贡献数

这个Hadoop任务仍然包括一个mapper和一个reducer。mapper非常简单，只是解析贡献者日志中的一行，并产生一个键值对，其键是贡献者的用户ID，其值是贡献记录的时间戳：

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java

```
public static class Map extends Mapper<Object, Text, IntWritable, LongWritable> {

  public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {

    Contribution contribution = new Contribution(value.toString());
    context.write(new IntWritable(contribution.contributorId),
      new LongWritable(contribution.timestamp));
  }
}
```

其中大部分工作都由Contribution类完成：

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/Contribution.java

```
Line 1 class Contribution {
     -   static final Pattern pattern = Pattern.compile("^([^\\s]*) (\\d*) (\\d*) (.*)$");
     -   static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();
     - 
     5   public long timestamp;
     -   public int id;
     -   public int contributorId;
     -   public String username;
     - 
    10   public Contribution(String line) {
     -     Matcher matcher = pattern.matcher(line);
     -     if(matcher.find()) {
     -       timestamp = isoFormat.parseDateTime(matcher.group(1)).getMillis();
     -       id = Integer.parseInt(matcher.group(2));
    15       contributorId = Integer.parseInt(matcher.group(3));
     -       username = matcher.group(4);
     -     }
     -   }
     - }
```

① http://dumps.wikimedia.org/enwiki

有很多种方法可以解析日志文件的一行——本例使用的是正则表达式（第2行）。如果其匹配，则使用Joda-Time库[1]的`ISODateTimeFormat`来解析时间戳，并将时间转换为长整数（第13行），这个长整数是自1970年1月1日到该时间所经过的毫秒数。贡献记录的ID和贡献者的用户ID是整数，这一行剩余的部分是贡献者的用户名。

reducer则会负责更多的工作：

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java

```
Line 1 public static class Reduce
     -   extends Reducer<IntWritable, LongWritable, IntWritable, Text> {
     -   static DateTimeFormatter dayFormat = ISODateTimeFormat.yearMonthDay();
     -   static DateTimeFormatter monthFormat = ISODateTimeFormat.yearMonth();
     5
     -   public void reduce(IntWritable key, Iterable<LongWritable> values,
     -                      Context context) throws IOException, InterruptedException {
     -     HashMap<DateTime, Integer> days = new HashMap<DateTime, Integer>();
     -     HashMap<DateTime, Integer> months = new HashMap<DateTime, Integer>();
    10     for (LongWritable value: values) {
     -       DateTime timestamp = new DateTime(value.get());
     -       DateTime day = timestamp.withTimeAtStartOfDay();
     -       DateTime month = day.withDayOfMonth(1);
     -       incrementCount(days, day);
    15       incrementCount(months, month);
     -     }
     -     for (Entry<DateTime, Integer> entry: days.entrySet())
     -       context.write(key, formatEntry(entry, dayFormat));
     -     for (Entry<DateTime, Integer> entry: months.entrySet())
    20       context.write(key, formatEntry(entry, monthFormat));
     -   }
     - }
```

这段代码首先为每个贡献者建立两个HashMap：`days`（第8行）和`months`（第9行）。然后遍历与这个贡献者相关的时间戳（`values`是时间戳列表），并用Joda-Time库的辅助方法`withTimeAtStartOfDay()`和`withDayOfMonth()`将时间戳转化为当天的午夜时间和当月第一天的午夜时间（分别是第12行和第13行）。接下来可以用一个简单的辅助方法对`days`和`months`的相关元素进行递增：

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java

```
private void incrementCount(HashMap<DateTime, Integer> counts, DateTime key) {
  Integer currentCount = counts.get(key);
  if (currentCount == null)
    counts.put(key, 1);
  else
    counts.put(key, currentCount + 1);
}
```

---

① http://www.joda.org/joda-time/

最后当两个HashMap构建完成，遍历这两个map就可以生成（至少有一条贡献记录的贡献者）每天和每月的贡献数（第17到20行）。

这里有一点需要说明，Hadoop任务的输出是键值对的集合，但本例中需要输出三个值——贡献者的用户ID、日期（某月或某天）和一个统计值。可以通过定义一个复合值来达到目的，键值对的键是贡献者的用户ID，而值是日期和统计值构成的复合值。不过由于本例非常简单，可以简单地用一个字符串作为值，这个字符串是通过`formatEntry()`定义的：

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java

```
private Text formatEntry(Entry<DateTime, Integer> entry,
                         DateTimeFormatter formatter) {
  return new Text(formatter.print(entry.getKey()) + "\t" + entry.getValue())
}
```

下面是这个任务的一部分输出：

```
463 2001-11-24 1
463 2002-02-14 1
463 2001-11-26 6
463 2001-10-01 1
463 2002-02 1
463 2001-10 1
463 2001-11 7
```

这就是我们想要的结果，但输出是一堆文本文件，不是很方便。下一节我们将学习服务层，其可以对批处理层的输出进行索引和合并。



小乔爱问：

是否可以增量地生成批处理视图？

到目前为止我们每次都是重新生成整个批处理视图。这是可行的，但也做了一些不必要的工作——为什么不用上次更新后的新数据增量地更新批处理视图呢？

没有什么能阻止我们这么做，而且这是一个非常有效的优化手段。不过需要提醒的是，我们不能严重依赖于增量更新——Lambda 架构的威力大部分来自于我们可以在必要时进行重建。所以只要值得优化就可以去实现一个增量算法，不过增量算法永远不能代替重建视图。

完成拼图

批处理层不能单独构成端到端的应用。所以还需要完成Lambda框架的另一部分——服务层。

服务层

我们需要对生成的批处理视图进行索引，这样就可以对索引进行查询了。另外，还需要一个

地方来存放程序逻辑（说明一个查询该如何合并批处理视图的逻辑）。这就是服务层的任务，如图8-8所示。

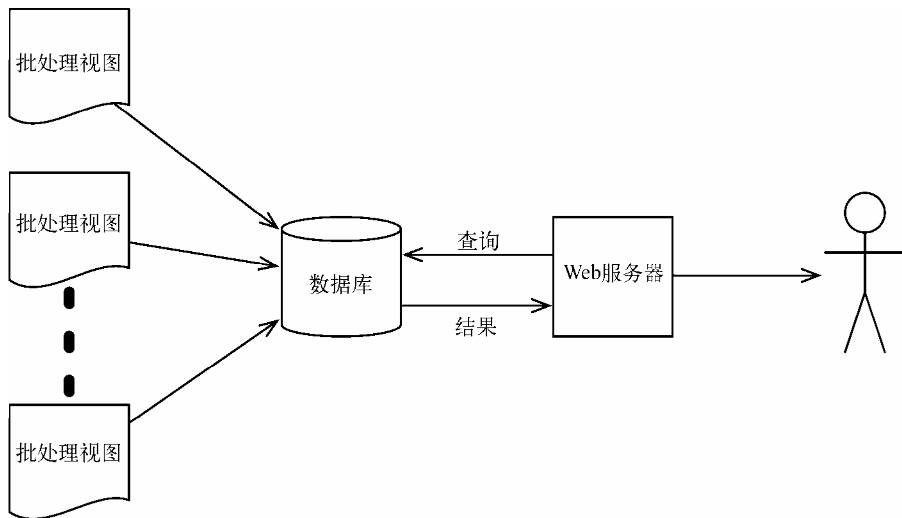


图8-8　批处理视图和服务层

由于服务层与本书主题关联度不大，对服务层的实现还是留作家庭作业。在此只介绍其中的一部分——数据库。

虽然我们可以利用传统数据库来构建服务层，不过其访问数据库的模式与传统应用不同。服务层不需要进行随机写——只需要在更新批处理视图时批量更新数据库即可。

有一类数据库为了这种访问模式而进行了优化，其中最有名的是ElephantDB[①]和Voldemort[②]。

涅槃

至此，利用批处理层和服务层，我们得到了一个数据系统，可以用于解决今天一开始提出的问题，如图8-9所示。

批处理层会不断循环运行，从原始数据重新生成批处理视图。每一个批处理完成时，服务层都会更新数据库。

由于只处理不可变的原始数据，批处理层可以轻松地被并行化。原始数据可以分布到一个多机集群，在可接受的时间内就可以处理TB级别的数据。

原始数据的不变性也使得系统可以承受住技术性故障和人为故障。一方面，原始数据更容易进行备份；另一方面，如果存在bug，最坏的情况就是批处理视图是暂时错误的——只需要修复

① https://github.com/nathanmarz/elephantdb
② http://www.project-voldemort.com/voldemort/

该bug并重新计算批处理视图即可。

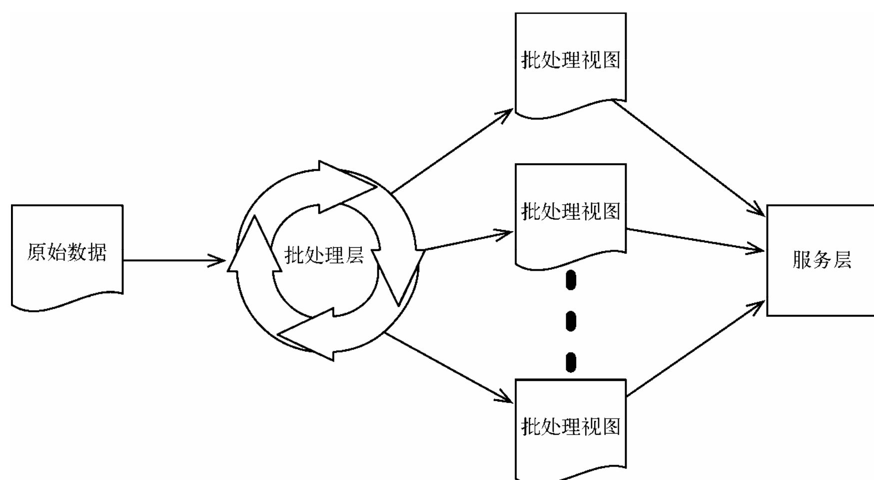


图8-9　数据系统

而且，由于保存了所有原始数据，就可以在将来生成任何想要的报表或进行任何分析。

不过存在着一个严重的问题——延迟。如果批处理层需要一个小时的运行时间，那批处理视图就比最新的数据至少延迟1小时。明天我们会学习加速层，来解决这个问题。

第二天总结

第二天的学习结束了。第三天我们将学习加速层，并完成整个Lambda架构。

第二天我们学到了什么

信息可以分为原始数据和衍生信息。原始数据是永恒的真相，而且是不变的。基于这个特性，利用Lambda架构的批处理层，可以创建具有以下特性的系统：

- 高度并行化，可以处理TB级别的数据；
- 简单，容易创建且不易出错；
- 对技术性故障和人为故障进行容错；
- 支持对日常数据的操作，也支持对历史数据生成报表和进行分析。

批处理层最大的缺点在于其有延迟，Lambda架构利用加速层来解决这一问题。

第二天自习

查找

- 本章中介绍的方法并不是利用Hadoop建立数据系统的唯一方法——其他的方法有HBase、

Pig和Hive。这三种方法更类似于传统数据系统。选择其中一种，与Lambda架构的批处理层进行比较。每种方法分别适合什么场景？

实践

- 创建一个服务层，来完善今天所学习的系统。这个服务层能够接受批处理层的输出，保存到数据库中，并可以查询一段时间内某用户的贡献数。可以使用传统数据库或ElephantDB来建立服务层。
- 扩展上面的例子，增量生成批处理视图——为了达到目的，Hadoop集群需要访问服务层的数据库。这个方案效率如何？花费的代价是否值得？增量生成批处理视图适用于何种应用？不适用于何种应用？

8.4　第三天：加速层

在昨天的学习中，我们了解到Lambda架构的批处理层能解决传统数据系统碰到的若干问题，但代价是较高的延迟。加速层就是用来解决这个问题的。图8-10展示了加速层与批处理层如何协同工作。

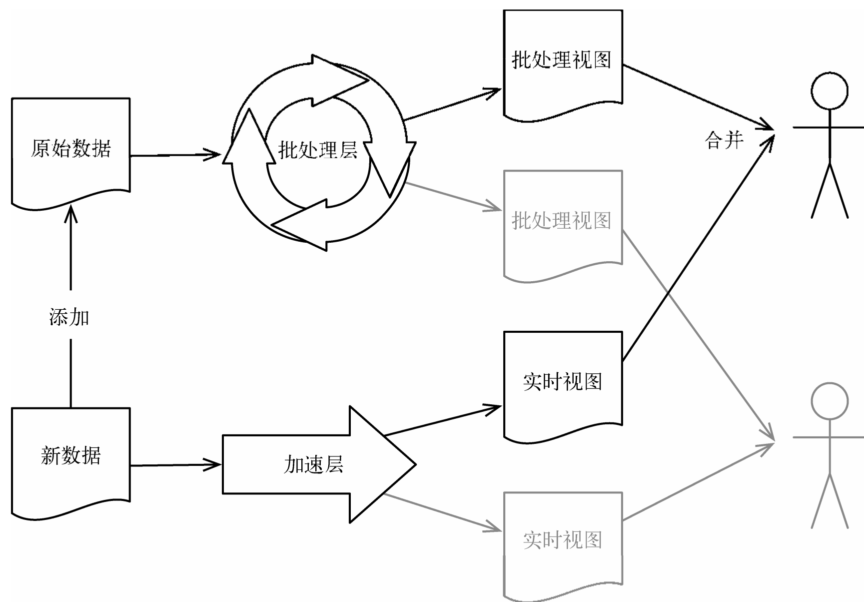


图8-10　Lambda架构

有新数据产生时，一方面将其添加到原始数据中，这样批处理层就可以进行处理；另一方面将其传给加速层。加速层会生成实时视图，实时视图会和批处理视图合并来满足对最新数据的查询。

实时视图仅包含最后一次生成批处理视图后产生的原始数据所对应的衍生信息，当这部分数据被批处理层处理后，该实时视图将被弃用。

现在我们用Storm①来生成加速层。

设计加速层

不同的应用对实时性的要求不同——有一些要求新数据在秒级别可用，有一些要求新数据在毫秒级别可用。无论你的应用有什么性能要求，只使用批处理层很可能无法满足。

由于加速层要求使用增量算法，因此比起构建批处理层，构建加速层本质上要更困难。这意味着加速层不能只处理原始数据，也就享受不到原始数据的完美特性了。我们必须重新面对传统数据库的特性：随机写、复杂的锁机制和事务机制等。

从好的方面来看，加速层只需要处理一部分数据，就是那部分还未被批处理层处理的数据（通常是几个小时的数据）。一旦批处理层赶上进度，旧的数据就会从加速层移除。

同步还是异步？

最容易想到的构建加速层的方法就是模仿传统的同步数据库。其实可以将传统数据库看作是Lambda架构的一种退化特例（不使用批处理层），如图8-11所示。

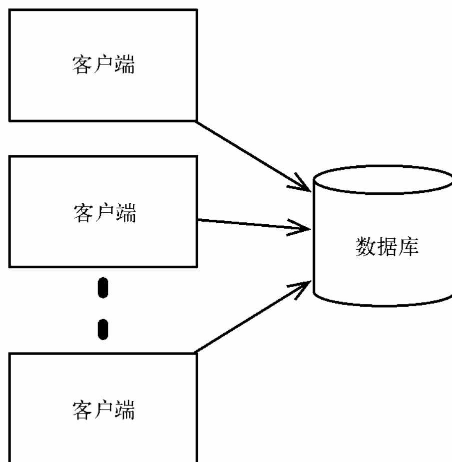


图8-11 传统数据库

在这种模型中，客户端直接和数据库通信，并在数据库进行更新操作时进行阻塞。这种模型非常合理，在某些场景下这是唯一能满足特定需求的方法。不过在另一些场景中，异步架构更合

① http://storm.incubator.apache.org

适一些，如图8-12所示。

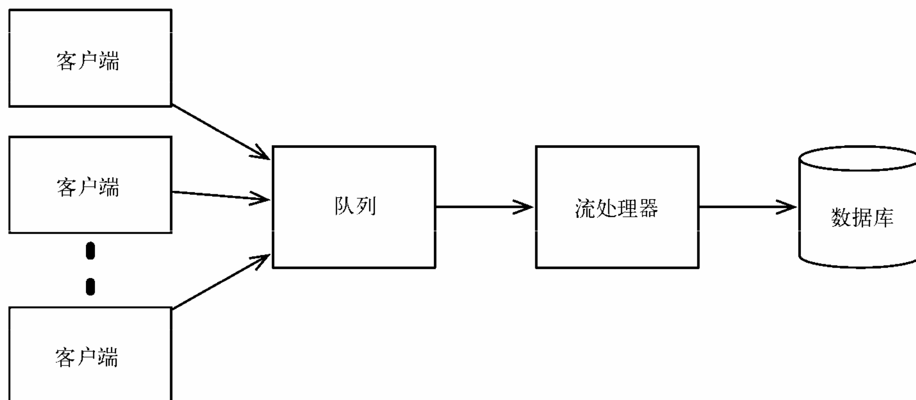


图8-12 传统数据库的异步架构

在这种模型中，客户端将更新操作添加到队列中（可以用Kafka[①]或Kestrel[②]等实现队列），这一步骤是无阻塞的。流处理器将串行地处理这些更新操作并对数据库进行更新。

用队列将客户端和数据库进行解耦，会使更新操作之间的交互变得更加复杂。不过，根据应用的特性，如果可以接受异步的方案，也会获得非常显著的好处：

- 客户端不会阻塞，所以少量的客户端就可以处理大量的数据，从而提高了吞吐量；
- 业务压力激增会导致客户端或数据库超载，也会导致同步系统超时或丢失一些更新。而异步系统则不同，只需要将未处理的更新操作保持在队列中，在业务压力恢复稳定后可逐渐赶上进度；
- 稍后我们将了解到：流处理器可以被并行化，也可以在多台计算机上进行分布式计算，既改善了性能又可以容错。

出于上述原因，再加上同步的加速层实现起来很是无趣，并且本书应关注于并行和并发，因此本书将不会深入讨论同步方案。在实现异步方案之前，需要先来学习如何让数据过期。

如何让数据过期

假设批处理层需要两个小时处理数据，那很容易就会认为加速层需要保留这两个小时以内的数据。实际上加速层需要保留两倍的数据，如图8-13所示。

假设第$N-1$次批处理刚刚结束，第N次批处理正要开始。如果每次批处理需要运行两个小时，

① http://kafka.apache.org
② http://robey.github.io/kestrel/

这意味着批处理视图会落后两个小时。因此加速层需要保持这落后的两个小时的数据，还要保持批处理层运行的这两个小时中所有的新数据，总共需要保持四个小时的数据。

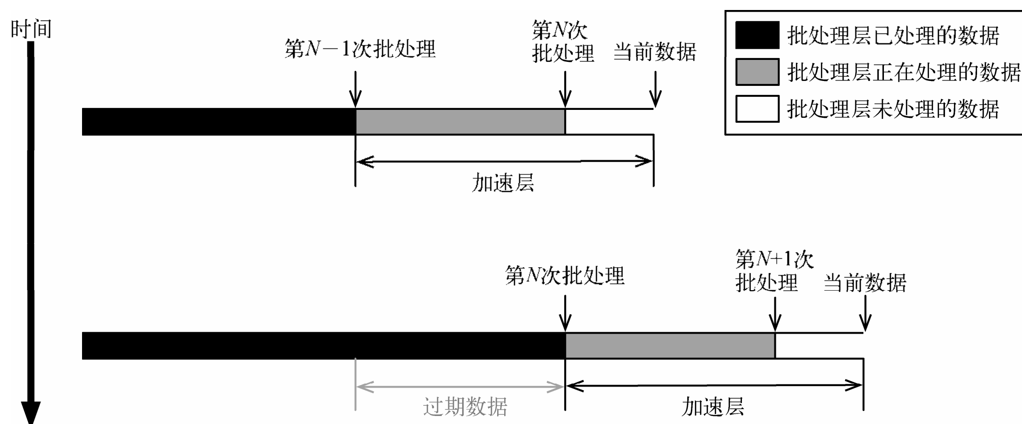


图8-13 在加速层中让数据过期

当第N次批处理结束时，需要让最早两个小时的数据过期，但仍保存其后两个小时的数据。有多种方法可以达到目的，不过最容易的就是同时维护两个加速层，并交替使用它们，如图8-14所示。

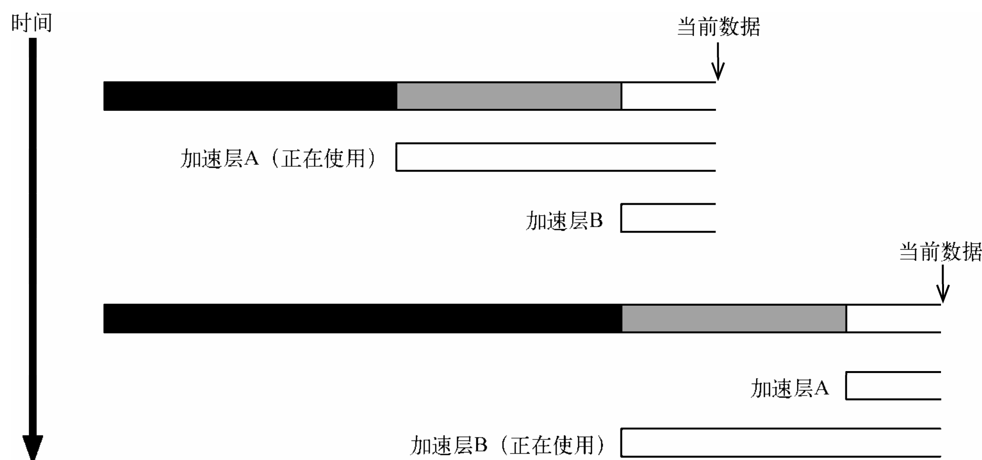


图8-14 交替使用加速层

当一次批处理完成时，批处理视图中的新数据就变得可用，就可以将当前用于处理请求的加速层切换到另一个加速层上。切换后闲置的加速层会清理其数据库，并在新的批处理开始时重新建立新的视图。

这种做法的好处是，一方面不需要费心识别加速层的数据库中哪些数据需要被过期清理，另一方面由于每次切换后加速层都是从一个空数据库开始运行，因此达到了更好的性能和可靠性。当然为此付出的代价是必须要维护两份加速层的数据并且消耗两份计算资源，不过考虑到加速层仅仅处理总数据量中很小的一部分，因此付出的代价相对不那么大。

Storm系统

剩下的时间我们来学习用Storm系统实现异步的加速层。Storm是个很大的话题，本书只能浅尝辄止——如需深入了解请参见Storm文档[①]。

Hadoop主要负责批处理，Storm主要负责实时处理——其能方便地使用多台计算机进行分布式计算，以改善性能和容错性。

Spout、Bolt和Topology

Storm系统处理的是元组（tuple）的流。Storm的元组类似于之前我们在第5章看到的actor模型的元组，但不同于Elixir的元组，Storm元组的元素是有名字的。

元组由spout（出水管）组件创建，并由bolt（螺栓）组件进行处理，bolt也会输出元组。用流将spout和bolt连接在一起，就形成了topology（拓扑结构）。图8-15所示的是一个简单的topology，由一个spout生成元组并由一系列bolt构成的流水线进行处理。

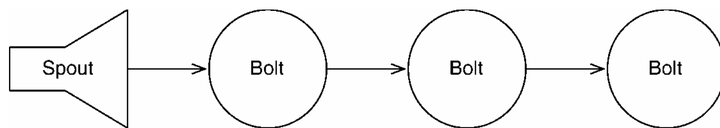


图8-15　一个简单的topology

topology也可以很复杂——bolt可以消费多个流，而一个流也可以被多个bolt消费，构成一个有向无环图（或称DAG），如图8-16所示。

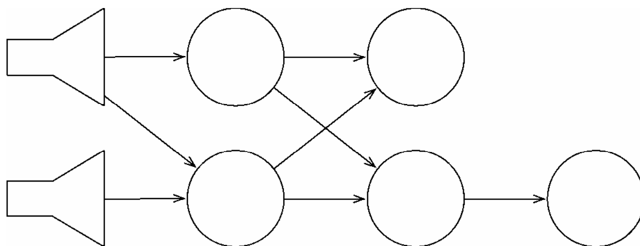


图8-16　一个复杂的topology

① http://storm.incubator.apache.org/documentation/Home.html

不过就算是最简单的那种流水线式的topology，也要比看上去的复杂很多，因为spout和bolt都是并行化和分布式的。

worker

spout和bolt不仅相互之间是并行的，而且其内部也都是并行的——每一个个体内部都是由很多worker实现的。如图8-17所示，这个简单的流水线式topology中，每个spout和bolt内部都有3个worker。

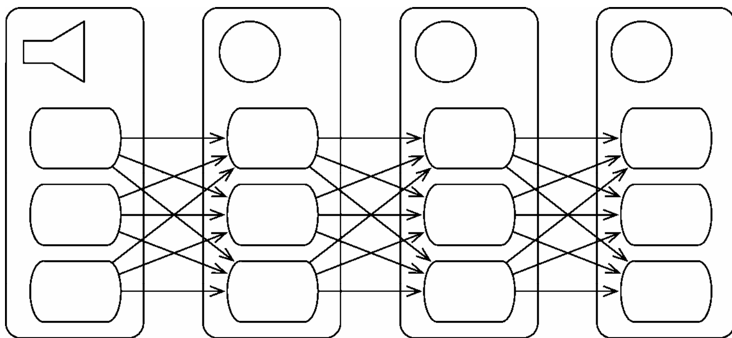


图8-17 spout和bolt的worker

如图8-17所示，流水线上每个节点的worker都可以向下游节点中任意一个worker发送元组。在稍后讨论到数据流分发（Stream Grouping）时我们会学习如何控制使用哪一个worker来接收元组。

worker还是分布式的——如果使用有4个节点的集群，那spout的worker可能运行在节点1、节点2和节点3上，第一个bolt的worker可能运行在节点2和节点4上（其中两个在节点2，一个在节点4上）。

Storm的优美之处在于我们不需要特别关注于分布式——只需要定义好topology，Storm就会向节点分配好worker，并确保发送的元组可以送达。

容错性

将一个spout或bolt的多个worker分布在多台计算机上的主要原因是容错性。如果集群中的某一台计算机发生故障，topology可以将元组分发给仍存活的计算机，这样topology就可以继续运行。

Storm会监视元组之间的依赖——如果某一个元组没能完成，Storm会将其依赖的spout元组置为失败并进行重试。这也就是说Storm默认使用的是“至少会执行一次”的处理策略。应用必须知道这个事实：元组可能会被重试，直到其结果正确。

听够了理论，我们来实践一下，为之前的Wikipedia贡献统计程序用Storm实现一个简单的加速层。



小乔爱问：

如果我的应用不能进行重试呢？

Storm 默认的策略是“至少会执行一次”，这一策略适用于大部分应用，但有一些应用需要更强的约束，即“只会执行一次”。

Storm 通过 Trident API[a] 可以支持“只会执行一次”的策略，本书将不会介绍相关内容。

a. http://storm.incubator.apache.org/documentation/Trident-tutorial.html

用Storm统计贡献

图8-18所示的是一种加速层的topology。

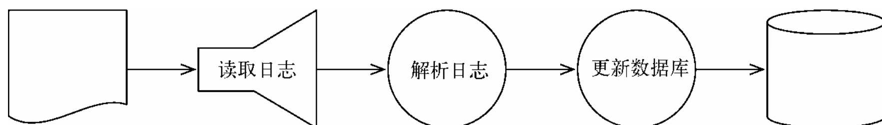


图8-18　加速层的topology

首先使用一个spout来读取贡献者的日志，并将其转换成一个元组流。然后由一个bolt来处理元组流，对日志条目进行解析，并输出一个解析后的流。最后再由另一个bolt处理这个流，并对保存实时视图的数据库进行更新。

不过出于以下原因，我们会构建一个不太一样的topology：首先，我们并不直接访问Wikipedia贡献者的日志；其次，我们并不想关心更新数据库的细节（我们关心的是并行和并发）。图8-19是我们设计的topology。

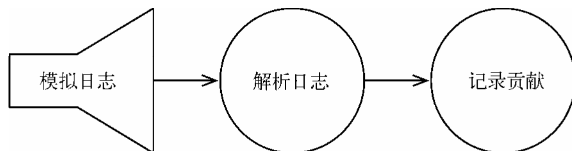


图8-19　改进后的加速层的topology

这个topology首先使用一个spout来模拟Wikipedia的贡献者feed，然后串联一个解析器，最后记录内存中的实时视图。这个方案不适用于产品环境，但非常适用于学习Storm。

模拟贡献日志

下面的代码实现了一个spout，它会随机产生日志来模拟feed：

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/RandomContributorSpout.java

```
Line 1 public class RandomContributorSpout extends BaseRichSpout {
     -
     -   private static final Random rand = new Random();
     -   private static final DateTimeFormatter isoFormat =
     5     ISODateTimeFormat.dateTimeNoMillis();
     -
     -   private SpoutOutputCollector collector;
     -   private int contributionId = 10000;
     -
    10   public void open(Map conf, TopologyContext context,
     -     SpoutOutputCollector collector) {
     -
     -     this.collector = collector;
     -   }
    15
     -   public void declareOutputFields(OutputFieldsDeclarer declarer) {
     -     declarer.declare(new Fields("line"));
     -   }
     -
    20   public void nextTuple() {
     -     Utils.sleep(rand.nextInt(100));
     -     ++contributionId;
     -     String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
     -       rand.nextInt(10000) + " " + "dummyusername";
    25     collector.emit(new Values(line));
     -   }
     - }
```

这段代码创建了一个spout，其继承了`BaseRichSpout`（第1行）。Storm会在初始化时调用`open()`方法（第10行）——在open方法中只是保存了`SpoutOutputCollector`的引用，以便之后将输出发给`SpoutOutputCollector`。Storm在初始化时还会调用`declareOutputFields()`方法（第16行），以便了解spout会产生的元组的结构——本例中，元组只有一个名为`line`的字段。

`nextTuple()`（第20行）承担了大部分工作。这个函数首先会随机睡眠100多毫秒，然后创建一个字符串，它的格式如8.3节的“贡献者日志”部分所述，最后调用`collector.emit()`来输出字符串。

产生的日志会被传给解析器bolt（将在下一节中介绍）。

解析日志

解析器bolt接受代表日志行的元组，并进行解析，再输出含有四个字段的元组，每个字段代表了日志行的一部分：

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/ContributionParser.java

```
Line 1 class ContributionParser extends BaseBasicBolt {
     2   public void declareOutputFields(OutputFieldsDeclarer declarer) {
     3     declarer.declare(new Fields("timestamp", "id", "contributorId", "username"));
     4   }
```

```
 5   public void execute(Tuple tuple, BasicOutputCollector collector) {
 6     Contribution contribution = new Contribution(tuple.getString(0));
 7     collector.emit(new Values(contribution.timestamp, contribution.id,
 8       contribution.contributorId, contribution.username));
 9   }
10 }
```

这段代码创建一个了bolt，其继承了`BaseBasicBolt`（第1行）。与创建spout时一样，还需要实现`declareOutputFields()`方法（第2行），以便让Storm了解到bolt输出元组的结构——本例中，输出元组包含4个字段，分别是`timestamp`、`id`、`contributorId`和`username`。

这次承担了大部分工作的是`execute()`（第5行）。本例与批处理层一样，使用了`Contribution`类来解析日志行，再调用`contributor.emit()`来输出元组。

解析得到的元组会被传给另一个bolt,以记录每个贡献者的贡献数,下一节我们会学习这个bolt。

记录贡献数

最后一个bolt维护了一个记录着每个贡献者的贡献记录的简单内存数据库（其实是一个map，其键是贡献者ID，其值是贡献时间戳的集合）：

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/ContributionRecord.java

```
Line 1 class ContributionRecord extends BaseBasicBolt {
     -   private static final HashMap<Integer, HashSet<Long>> timestamps =
     -     new HashMap<Integer, HashSet<Long>>();
     - 
     5   public void declareOutputFields(OutputFieldsDeclarer declarer) {
     -   }
     -   public void execute(Tuple tuple, BasicOutputCollector collector) {
     -     addTimestamp(tuple.getInteger(2), tuple.getLong(0));
     -   }
    10 
     -   private void addTimestamp(int contributorId, long timestamp) {
     -     HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
     -     if (contributorTimestamps == null) {
     -       contributorTimestamps = new HashSet<Long>();
    15       timestamps.put(contributorId, contributorTimestamps);
     -     }
     -     contributorTimestamps.add(timestamp);
     -   }
     - }
```

本例中并不产生任何输出，所以`declareOutputFields()`函数是空的（第5行）。`execute()`（第7行）方法只是从输入中提取相关信息，并传给`addTimestamp()`函数，`addTimestamp()`负责向贡献者相应的集合中添加时间戳。

现在来创建一个topology，将已有的spout和bolt集成起来。



小乔爱问：

为什么要记录时间戳的集合？

在8.3节中我们看到批处理视图仅记录了每天和每月的贡献记录数。那么在实时视图中为什么要记录完整的时间戳呢？

如之前讨论过的，实时视图只需要记录几个小时的数据，所以记录和查询完整的时间戳的代价相对较低。但更重要的原因是：向集合中增加记录的操作是幂等的。

之前学习过，Storm默认的策略是“至少会执行一次”，那么元组可能会被重试。所谓幂等操作，就是无论操作执行多少次结果都是一样的，这正可以用于处理元组被重试的情况。

构建topology

我们对ContributionRecord可能会有些担心——已经知道bolt会包括多个worker，那么如何来保证一个贡献者只会对应一个时间戳集合呢？要解决这个问题就需要先了解如何构建topology。

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/WikiContributorsTopology.java

```
Line 1 public class WikiContributorsTopology {
     -
     -   public static void main(String[] args) throws Exception {
     -
     5     TopologyBuilder builder = new TopologyBuilder();
     -
     -     builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);
     -
     -     builder.setBolt("contribution_parser", new ContributionParser(), 4).
    10       shuffleGrouping("contribution_spout");
     -
     -     builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
     -       fieldsGrouping("contribution_parser", new Fields("contributorId"));
     -
    15     LocalCluster cluster = new LocalCluster();
     -     Config conf = new Config();
     -     cluster.submitTopology("wiki-contributors", conf, builder.createTopology());
     -
     -     Thread.sleep(10000);
    20
     -     cluster.shutdown();
     -   }
     - }
```

首先，这段代码创建一个TopologyBuilder（第5行），并调用setSpout()（第7行）来添加一个spout实例。其第二个参数是一个并行度的参考（hint），这只是个参考而不是强制命令，但

为了理解简单，可以认为这个参数是一个强制命令，Storm会根据这个参数为spout创建4个worker。如果要详细了解如何控制Storm的并行，推荐阅读“What Makes a Running Topology: Worker Processes, Executors and Tasks”[1]。

接下来，这段代码调用`setBolt()`（第9行）来添加`ContributionParser`的实例。最后，这段代码调用`shuffleGrouping()`，其参数与设置spout时的字符串一样，这样Storm就知道这个bolt需要从spout中读取输入。这里我们还需要学习数据流分发。

数据流分发

Storm的*数据流分发*策略主要解决了哪一个worker接受哪一个元组的问题。解析器bolt所使用的*随机分发*（shuffle grouping）策略是最简单的——只是简单地将元组随机分发给某一个worker。

记录贡献数的bolt使用的是*按字段分发*（fields grouping）策略（第12行），这个策略保证某些字段（本例中是`contributorId`字段）的值相同的元组会被分发给同一个worker。回到上一节开始的问题，我们也是通过这个策略来确保一个贡献者只对应一个时间戳集合的。

本地集群

设置Storm集群并不复杂，但这超出了本书的范围。更悲剧的是，由于这是一门新技术，还没有可以直接利用的现成的Storm集群服务。所以需要创建了一个`LocalCluster`（第17行）在本地运行我们的topology。

本例中我们让topology运行了10秒后调用`cluster.shutdown()`来关闭它。然而在产品环境中，当批处理层赶上了进度，已经不再需要当前的实时视图时，就需要用其他方法来关闭topology。

第三天总结

我们完成了第三天的学习，也完成了对Lambda架构的加速层的讨论。

第三天我们学到了什么

加速层创建的实时视图包含了最后一次生成批处理视图后产生的数据，这样就完善了Lambda架构。加速层可以是同步的或异步的——Storm是一种构建异步加速层的方法。

- Storm实时地处理元组流。元组由spout创建、由bolt处理、由topology调度。
- spout和bolt都包含多个worker，这些worker并行执行，且分布在集群的多个节点上。
- Storm默认使用“至少会执行一次”的策略——bolt需要处理元组被重试的情况。

---

① http://storm.incubator.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html

第三天自习

查找

- Trident是建立在Storm基础上的高级API。类似于Storm的“至少会执行一次”的默认策略，Trident提供了“只会执行一次”的策略。Trident适用于什么场景？Storm的低级API适用于什么场景？
- 除了随机分发和按字段分发，Storm还提供了哪些数据流分发策略？

实践

- 创建一个真正的Storm集群，并将今天本地运行的例子部署在上面。
- 创建一个bolt和一个topology，bolt负责维护每分钟的贡献总数，topology负责将`ContributionParser`的输出分发给`ContributionRecord`和我们创建的bolt。
- 今天的例子使用了`BaseBasicBolt`，它会自动地对元组进行确认。请改用`BaseRichBolt`——你需要显式对元组进行确认。创建一个bolt，如何在确认之前处理多个元组？

8.5 复习

Lambda架构将我们已经学过的一些内容融合在了一起：

- 原始数据是永恒的真相，这让我们想到Clojure分离标识与状态的做法；
- Hadoop并行化解决问题的方法是先将问题切分并映射到一个数据结构上，再进行化简操作。这非常类似于并行函数编程的做法；
- 类似于actor模型，Lambda架构将处理过程分布到集群上，这样既改进了性能，又可以对硬件故障进行容错；
- Storm的元组流类似于actor模型和CSP模型的消息机制。

优点

Lambda架构主要用于解决大规模数据的问题——这些问题是传统数据处理架构难以应对的。Lambda架构非常适合于报表和分析——以前我们会使用数据仓库来进行这类工作。

缺点

Lambda架构最大的优点——擅长处理大规模数据——这也正是它的缺点。除非你的数据达到数太字节甚至更多，否则其成本（计算成本和智力成本）将高于收益。

替代方案

Lambda架构并没有与MapReduce绑定——批处理层可以用其他的分布式批处理系统来实现。

基于这一点，Apache Spark[1]就是一个很有意思的方案。Spark是一个集群计算框架，它实现了DAG执行引擎，可以使用很多比MapReduce用起来更自然的算法（尤其是图算法）。它也提供了与流相关的API[2]，这意味着批处理层和加速层都可以用Spark实现。

结语

由于包含前几章介绍过的很多技术，Lambda架构很适合用来作为本书压轴的主题。用Lambda架构演示“如何利用并行和并发技术解决一些棘手问题”是非常合适的。

最后一章，我们将重新审视过去7周的内容，以及本书已经涉及的几大主题。

① http://spark.apache.org
② http://spark.apache.org/streaming/

第 9 章 圆满结束

9

恭喜你完成了七周的学习！

从由数据并行GPU提供的细粒度并行，到大规模的MapReduce集群，我们讨论的主题涉及方方面面。一路走来，我们不仅学习了如何用并行和并发来挖掘现代多核CPU的潜力，而且学到了许多比传统串行代码更优秀的特性。

- 我们学习了Elixir、Hadoop和Storm，这几种系统都可以部署在多机集群上进行分布式计算，从而创建出可以对硬件故障进行容错的解决方案。
- 通过`core.async`，学习了如何利用并发来解决事件处理时会碰到的“回调困境”。
- 在函数式编程的章节中，学习了如何让并发方案比等价的串行方案更简洁易读。

现在来看看这预示着怎样的未来。

9.1 君欲何往

二十几年前，我曾预言并行技术和分布式编程将成为主流，如今的现实表明我不是个成功的预言家。尽管如此，现在我仍然相信并行和并发预示着编程的未来。

未来是“不变”的

在我看来，有一个话题散发着耀眼的光芒——和过去相比，不变性在代码中的应用会越来越广泛。与不变性关系最大的是函数式编程——在函数式编程中，避免使用可变状态会使得并行和并发更为简单。不过为了获得不变性，不一定非要使用函数式编程。在过去的几周中我们已经学过：

- 虽然Clojure不是一门纯粹的函数式语言，但其核心数据结构是持久且不变的（参见4.2节的“持久数据结构”部分）。持久数据结构可以将标识与状态分离，这样Clojure就可以支持可变引用，并且避免使用可变状态带来的问题；
- 虽然Lambda架构的底层代码通常不是函数式代码，不过其核心思想的确是不变性——批

处理层规定其原始数据是永恒真实的（不可变的），所以我们可以将数据安全地分布到集群中，对数据进行并行处理，且能对技术故障和人为故障进行容错；
- 运行在Erlang虚拟机上的Elixir有着卓越性能和可靠性，虽然它不是一门纯粹的函数式语言，但其优异表现的关键是它不适用可变变量；
- 基于actor模型或CSP模型的应用所发送的消息是不可变的；
- 在使用线程与锁模型时，不变性也非常有用——不可变的数据越多，需要使用的锁就越少，我们就越不用担心内存可见性带来的问题。

显而易见，虽然我们可能不使用函数式语言，但所涉及的框架和代码越来越多地受到函数式规则的影响。这是个好消息——不仅让我们更容易地使用并行和并发，也让代码变得更加简洁易懂且可靠。

未来是分布式的

并行和并发目前的复兴主要是由多核危机引发的。CPU的发展趋势并不是大幅提升单核性能，而是增加CPU的核数。好消息是利用过去几周所学到的知识我们可以挖掘出多核的潜力。

但我们还面临着另一个危机——内存带宽。目前，双核、4核或8核的计算机尚可利用共享内存高效地通信，但如果涉及16核、32核甚至64核呢？

如果CPU的核数继续按照这个速度增长，共享内存就会成为瓶颈，分布式内存就成为我们不可不考虑的选择。未来的计算机可能仍然是个小盒子，但从程序员的角度来看，其更像一个计算机集群。

我认为，基于消息传递的技术，例如actor模型和CSP模型，随着时间发展会变得愈加重要。

你肯定猜到了，过去的七周内我们没有穷尽并行和并发的每一种可能。那我们遗漏了些什么呢？

9.2　未尽之路

撰写本书时，最艰难的就是对内容进行取舍。下面简要介绍一些我们尚未涉及的技术，以及一些自学的资料。

Fork/Join模型和Work-Stealing算法

Fork/Join是随着Cilk语言[1]流行起来的并行方法，Cilk是C/C++的一个并行变种。不过现在许多语言环境，包括Java[2]，都实现了Fork/Join。Fork/Join模型非常适用于分治算法，我们在3.3节

① http://www.cilkplus.org

② http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

的“分而治之”部分学习过分治算法（实际上Clojure的reducer内部就使用了Fork/Join模型）。

实现Fork/Join，通常会用到work-stealing算法在线程池中共享任务。work-stealing非常类似于Clojure中的go块（参见6.2节的“go块”部分）。

数据流

我们在3.4节中接触过数据流，这个主题值得更深入的讨论。本书之所以未作深入讨论的主要原因是，没有找到一门合适的、通用的数据流语言。较为合适的语言是多重编程范式语言Oz[①]（Mozart编程系统的一部分）。

本书不深入讨论并不意味着数据流不重要——恰恰相反，硬件设计中大量使用了基于数据流的并行技术——VHDL[②]和Verilog[③]都是数据流语言。

反应型编程

与数据流密切相关的是反应型编程（reactive programming）。反应型程序可以自动传播变化。反应型程序之所以引人关注，归功于Microsoft Rx（Reactive Extensions）库[④]和其他的库[⑤]。

反应型编程与之前我们学习的几种技术有相似之处，包括Storm的topology，还有actor模型和CSP模型这类基于消息传递的技术。

函数式反应型编程

函数式反应型编程（Functional Reactive Programming，FRP）是反应型编程的一种，通过对时间进行建模来扩展函数式编程。Elm[⑥]实现了并发版本的FRP，其运行在浏览器中。与`core.async`类似，在处理事件时其提供了一种方法来避免“回调困境”。Elm是本系列丛书的下一本书（*Seven More Languages in Seven Weeks* [TDMD14]）中将要涉及的编程语言之一。

网格计算

网格计算是一种松耦合地建立分布式集群的方法。网格的元素通常是异构且地理分布的，甚至加入网格和退出网格都可能是自发的。

---

① http://mozart.github.io
② http://en.wikipedia.org/wiki/VHDL
③ http://en.wikipedia.org/wiki/Verilog
④ https://rx.codeplex.com
⑤ https://github.com/Netflix/RxJava
⑥ http://elm-lang.org

最著名的网格计算项目是SETI@Home[1]，任何人都可以通过它参与到许多项目的计算中。

元组空间

元组空间（tuple space）是分布式联想记忆（distributed associative memory）的一种形式，可用于实现进程之间的通信。元组空间首次在Linda协作语言[2]中被引入（这恰巧是20世纪90年代初我的博士论文选题），现在也有一些正在开发中的基于元组空间模型的系统[3,4]。

9.3　越过山丘

我是一个汽车迷，所以每一章开头使用的比喻几乎都与汽车相关。与汽车类似，编程遇到的问题会呈现不同的类型和不同的规模。无论我们处理的问题相当于一辆轻量级定制赛车、一辆量产家用轿车，或者一辆重型卡车，我都可以自信地说，并行和并发都将变得越来越重要。

无论你是否会直接使用这些并发模型，我真诚地希望过去七周所学的知识能帮助你更有信心地迎接未来的项目。祝驾驶（线程）安全。

---

① http://setiathome.ssl.berkeley.edu
② http://en.wikipedia.org/wiki/Linda_(coordination_language)
③ http://river.apache.org/
④ https://github.com/vjoel/tupelo

参考书目

[Arm13] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, Second, 2013.

[Goe06] Brian Goetz. *Java Concurrency in Practice*. Addison-Wesley, Reading, MA,2006.

[HB12] Stuart Halloway and Aaron Bedra. *Programming Clojure*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, Second, 2012.

[MD14] Jack Moffitt and Fred Daoud. *Seven Web Frameworks in Seven Weeks:Adventures in Better Web Apps*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2014.

[MW14] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich,CT, 2014.

[RW12] Eric Redmond and Jim R. Wilson. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2012.

[Tat10] Bruce A. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. The Pragmatic Bookshelf, Raleigh, NCand Dallas, TX, 2010.

[TDMD14] Bruce A. Tate, Fred Daoud, Jack Moffitt, and Ian Dees. *Seven More Languages in Seven Weeks*. The Pragmatic Bookshelf, Raleigh, NC and Dallas,TX, 2014.

[Tho14] David Thomas. *Programming Elixir: Functional |> Concurrent |> Pragmatic|> Fun*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2014.

版 权 声 明

“基于锁和线程的并发模型是目前最常用的一种并发模型，但是并发编程模型不仅仅只有这一种，本书几乎涵盖了目前所有的并发编程模型。了解和熟悉各种并发编程模型，在解决并发问题时会有更多思路。”

——方腾飞，并发编程网站长

“当看到这本书的目录时，我就为之一振。它涉及了当今所有的主流并发编程模型（当然也包括Go语言及其实现的CSP模型）。而在品读之后，我更是有一些激动。其中的大部分章节都深入浅出地描述了某一个并发编程模型的概念、特点、机理以及适用领域。加之有大量相应实现语言的示例呈现，更是让这些模型活灵活现、极易理解。另外，各章之间的衔接会使你更容易把这些知识融会贯通。总之，并发编程乃现代编程技能的必备，而这本书则为并发编程的首选佳作。”

——郝林（@特价萝卜），《Go并发编程实战》作者

“几十年来，专业的程序员一直在用线程与锁模型来进行并行和并发，但这仅仅是《七周七并发模型》介绍的众多模型之一。当今主流语言都竞相支持不同的并发模型，例如CSP模型、数据并行、函数式编程和Clojure的unified succession model。如果你想紧跟时代，我推荐阅读本书。”

——Stuart Halloway，Cognitect联合创始人

“随着计算机拥有越来越多的核，理解并发技术变得愈发重要。本书介绍了为什么函数式编程会与并发有关、如何利用actor模型编写分布式软件、怎样利用GPU实现并行处理以及大数据的相关内容。通过阅读本书，你可以充实自己的知识库，为未来数年做好准备。”

——José Valim，Plataformatec联合创始人

“这是个变革的时代，每个程序员都应学会并发编程的思维方式。如果你问我‘应当如何适应这个并发的时代’，我会推荐阅读本书，因为我从中受益良多。”

——Andrew Haley，Red Hat公司首席Java工程师

“随着阿姆达尔定律开始取代摩尔定律的地位，面向并发编程也逐渐开始取代面向对象编程的地位。本书的问世恰逢其时。Paul在书中详尽介绍了最重要的几种并发模型，足以帮助读者选择最适合的并发模型。如果你要开发与多核相关的软件，我推荐必读本书。”

——Francesco Cesarini，Erlang Solutions创始人、技术总监

“我与Paul有过多次合作，我确信他是编程语言设计和结构方面的权威。并发是现代软件工程上非常重要但又容易被误读的主题，本书清晰阐述了这个主题。”

——Ben Medlock，SwiftKey联合创始人、CTO

The Pragmatic Programmers

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-38606-9

定价：49.00元